RL-TR-97-240
Final Technical Report
March 1998

REAL-TIME SOFTWARE VISUALIZATION

Kestrel Institute

Cordell Green, Rafael Furst, Jim McDonald, and Stephen Westfold

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.

AIR FORCE RESEARCH LABORATORY
ROME RESEARCH SITE
ROME, NEW YORK

This report has been reviewed by the Air Force Research Laboratory, Information Directorate, Public Affairs Office (IFOIPA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

RL-TR-97-240 has been reviewed and is approved for publication.

APPROVED:

MARK J. GERKEN, Capt, USAF
Project Engineer

FOR THE DIRECTOR:

WARREN H. DEBANY, JR., Technical Advisor
Command, Control, & Communications Directorate

If your address has changed or if you wish to be removed from the Air Force Research Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify AFRL/IFTD, 525 Brooks Road, Rome, NY 13441-4505. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

ALTHOUGH THIS REPORT IS BEING PUBLISHED BY AFRL, THE RESEARCH WAS ACCOMPLISHED BY THE FORMER ROME LABORATORY AND, AS SUCH, APPROVAL SIGNATURES/TITLES REFLECT APPROPRIATE AUTHORITY FOR PUBLICATION AT THAT TIME.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY *(Leave blank)*	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	March 1998	Final Sep 93 - Mar 96

4. TITLE AND SUBTITLE

REAL-TIME SOFTWARE VISUALIZATION

5. FUNDING NUMBERS

C - F30602-93-C-0236
PE - 62702F
PR - 5581
TA - 27
WU - 73

6. AUTHOR(S)

Cordell Green, Rafael Furst, Jim McDonald, and Stephen Westfold

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304

8. PERFORMING ORGANIZATION REPORT NUMBER

N/A

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

AFRL/IFTD
525 Brooks Road
Rome, NY 13441-4505

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

RL-TR-97-240

11. SUPPLEMENTARY NOTES

AFRL Project Engineer: Capt Mark J. Gerken/IFTD/(315) 330-2974

12a. DISTRIBUTION AVAILABILITY STATEMENT

Approved for public release; distribution unlimited.

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

This report describes a formal approach to visualizing software specifications. The approach taken was to begin with a formal description of the entity to be visualized and repeatedly apply automated inference steps to transform the description into a form which could be rendered using standard techniques. The Specware specification construction environment was used to build specifications defining these visualizations. Several examples are included. In addition to visual representation, the role of sound in understanding and visualizing complex structures was also investigated.

14. SUBJECT TERMS

software visualization, formal methods, knowledge-base

15. NUMBER OF PAGES

80

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL

Table of Contents

1. Overview

Traditionally, visualization of software has been ineffective for the development cycle due to the long turnaround time to create visualizations that are **relevant** to both the level of abstraction and domain-specificity that the software developer is working with at any given time. The "post-mortem" approach of creating visualizations by hand after development of, say a data structure, necessitates a loss of time and money when the data structure changes and the visualization has to be redone. Also, errors could be introduced in such a manual visualization process. We believe that the Knowledge-Based Software Assistant (KBSA) "machine-in-the-loop" approach [GLB+86,GRW96] can be extended to the area of software visualization, and thereby remove the stumbling blocks of slow turnaround time and human error. Seen in the KBSA light, visualization is not a process that occurs "after the fact." Rather, like validation, visualization is co-derived with software and maintained similarly.

We have done research and have implemented an experimental prototype system to meet the goal of automatically producing visualizations of software entities, such as abstract data types. The approach we have taken is analogous to the approach we have taken in the past for automated derivation of algorithms from specifications: begin with a formal description of the entity to be visualized, and repeatedly apply automated inference steps to transform the description into a form which can be rendered via standard techniques. The implementation is being created using our latest system for software design and development, called Specware, built using KBSA technology.[JS93,JS95]

In this report, we document our experiences using Specware to construct formal specifications for data visualization. This report is organized as follows:

- Section 2 provides a conceptual overview of the VIZO system;
- Section 3 provides an overview of the VIZO system;
- Section 4 describes a sample visualization;
- Section 5 describes the VIZO system in greater detail;
- Section 6 describes the role of sound in vizualization;
- Section 7 contains a summary of this research;
- Appendix A contains a more detailed example; and
- Appendix B describes the Specware system in greater detail.

2. Visualization via Abstract Views Enables Automation

One of the roadblocks that most visualization systems encounter in achieving automation is the variety of different types of entities that may potentially be represented crossed with the variety of potential ways to represent them. This leads to one of two approaches: either attempt to pre-anticipate as many solutions as possible (which inevitably restricts the range of applicability to a narrow domain), or require the human to do the work of visualization with a set of base tools provided by the system.

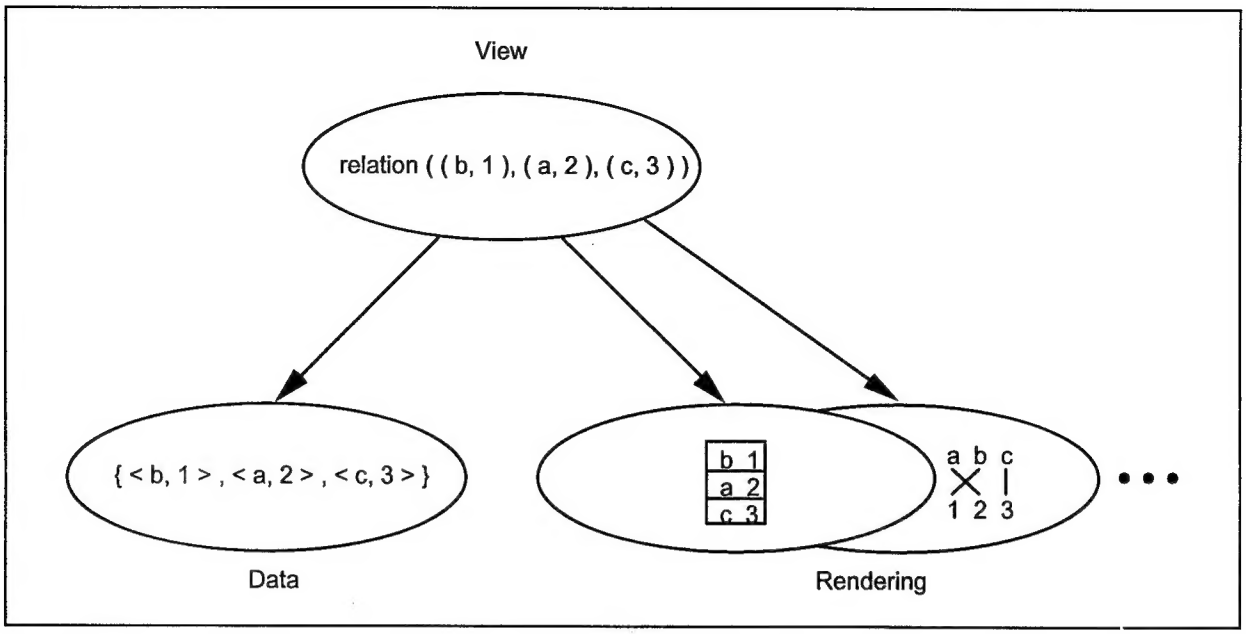


Figure 1. Relation between View, Rendering and Data

Our approach begins with the separation of the visualization process into two steps. First, categorize the entity being visualized (call this the "data" for convenience, though it may in general be a process which will become animated) as an instance of an abstract "view" or "pattern". Then, using the inference process described in Section 3, transform the view into a "rendering" which reflects the properties of the view. Figure 1 shows the relationship between the data, view and rendering for a small abstract datatype (a set of pairs). In the case of Figure 1, the entity being visualized is categorized as a set of tuples and two renderings are shown for these tuples.

Notice that the same view can have multiple renderings, and although it is not depicted, the same data may have different views. Views can be combined and manipulated automatically because they are formally described (i.e., their properties are made explicit) in the logical framework of Specware. This approach reduces the necessary pre-anticipated constructs to the small set of independent visualization concepts needed to make a complex visualization (e.g. adjacency, containment), enabling automation to take place. Figure 2 shows a partial hierarchy of aggregate views which can be built using the more basic views.

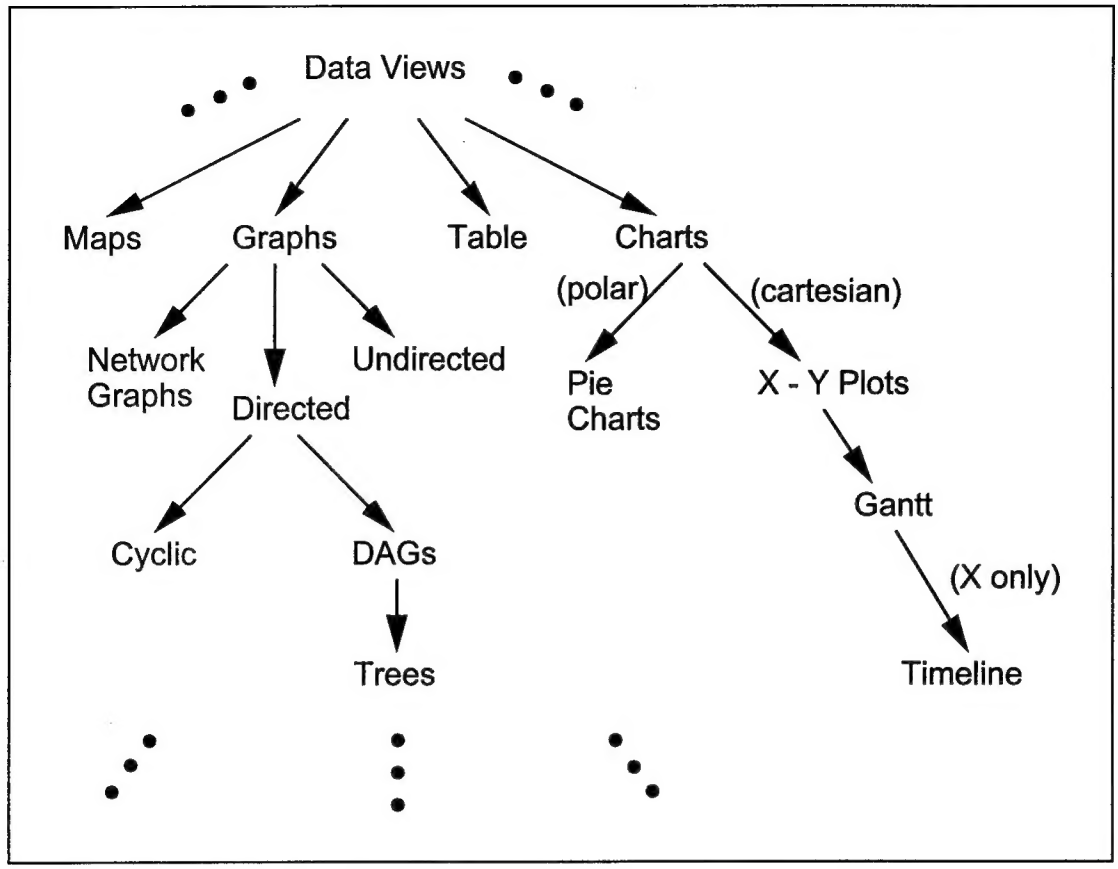


Figure 2. Abstract View Hierarchy for Visualization

Hierarchies such as this are common in the Specware framework, and enable the incremental refinement of one view into another (working down the hierarchy), eventually ending up with a "renderable" description of the desired data. Once automated, a human can then choose between different alternative renderings of the data, depending on the task at hand. Novel visualizations result from the combination of views that previously have not been combined. For example, Gantt and DAG views can be combined to form hybrid dependency/timing diagrams.

3. VIZO Architecture: Overview

The prototype VIZO system attempts to realize the preceding model of automatic visualization construction. The VIZO architecture, seen below, shows the visualization process at a high-level overview.

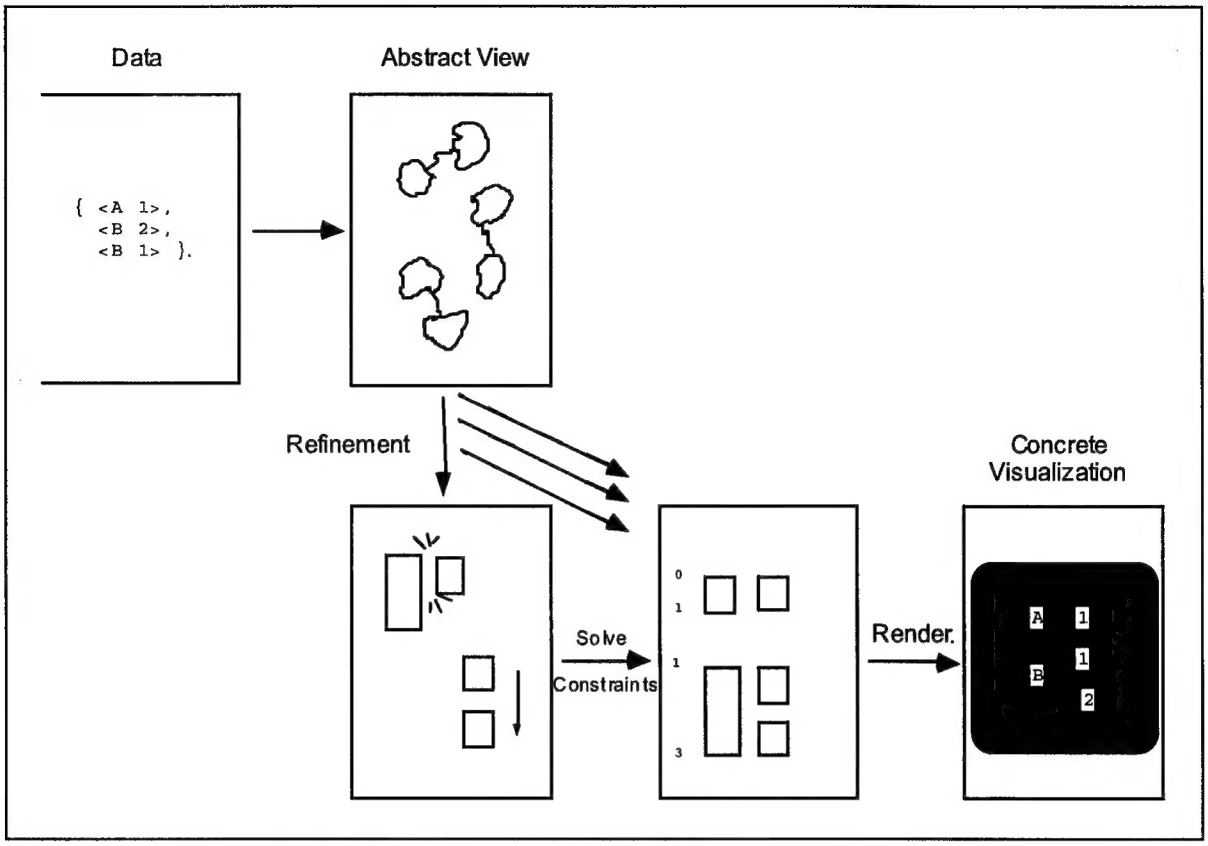


Figure 3. VIZO Architecture

Notice the "data" and abstract "view" in the two boxes in the upper-left part of the diagram map directly to the data and view from Figure 3. The final "rendering" appears in the box in the lower-right. The boxes and arrows between the view and the rendering depict the automated reasoning process which results in a visual representation of the data. In VIZO, two types of automated reasoning occur. The first is called *refinement*, a process that is based on an automated theorem proving engine and is directly supported in Specware. The second is the use of specialized constraint solvers. Both of these processes and the roles they play are discussed below.

3.1. Refinement

Refinement is a formal operation in Specware in which an *interpretation* is created from one view to another, allowing any data which can be described using the first view to also be described using the second view. Typically, interpretations are used to map abstract entities into concrete implementations, such as mapping the sorts and operations of a Set abstract data type (ADT) into the sorts and operations of a List ADT. In the visualization domain, given a view which describes the data abstractly as a related pair of regions (on the screen), there is an interpretation which maps this view into a more refined view in

which pairs of related regions become ovals connected with a line between them. In this example the concept of an amorphous "region" is refined to a specialized instance of a region, namely an oval. Similarly, the concept of two regions being "related" is refined to a specialized representation of the relation, namely being connected with a line. An alternate refinement for the same original view is that of a pair of adjacent boxes. Figure 4 shows how refinements are used to transform a set of pairs of numbers into a table which can be rendered on the screen.

Notice that in several places (i.e. the two ordering steps and the two adjacency steps), existing refinements were used in different contexts. This is one example of how our approach eliminates the need to pre-anticipate solutions in order to cover the space of useful visualizations.

3.2. Constraint Solving

The result of the refinement process is a formal description that, given a graphics system with primitive operations that match those in the description, can be directly rendered. Often times it is desirable that the output of the refinement leave certain layout decisions uncommitted, allowing multiple solutions which are consistent with the current view. For instance, the view may specify that two objects be aligned on their top boundary, but it may not matter how close they are in the horizontal dimension. In such a case there are many possible final renderings which are consistent with the current view. The remaining task is to determine where to place both objects on the screen subject to the constraint that the top boundaries align (and possibly subject to other constraints as well). Here, we employ a constraint solver to determine a consistent or optimal solution. Figure 5 shows several of the constraints which commonly arise in visualization, depicted by example.

Constraint solvers are automated reasoning engines, similar to automated theorem provers, though they are usually tailored to a specific domain of discourse (such as integer linear systems of equations). Because of this, they can be made very efficient for the class of problems they address. The match between the kinds of constraints we need to solve in VIZO and the traditional domains of constraint solvers (i.e. numerical) is a good one. We have explored several freely available constraint solvers: SkyBlue, Omega, and an implementation of a solver described in Guy Steele's doctoral dissertation. Omega is currently integrated into our theorem prover. While each of these systems may be made to work with the VIZO system, we feel that better constraint-solving technology – one handling more complex and aggregate constraints -- will greatly aid the system, and furthermore, such technology is likely to be built within the next few years as the importance of constraint-based reasoning is being recognized in the larger community.

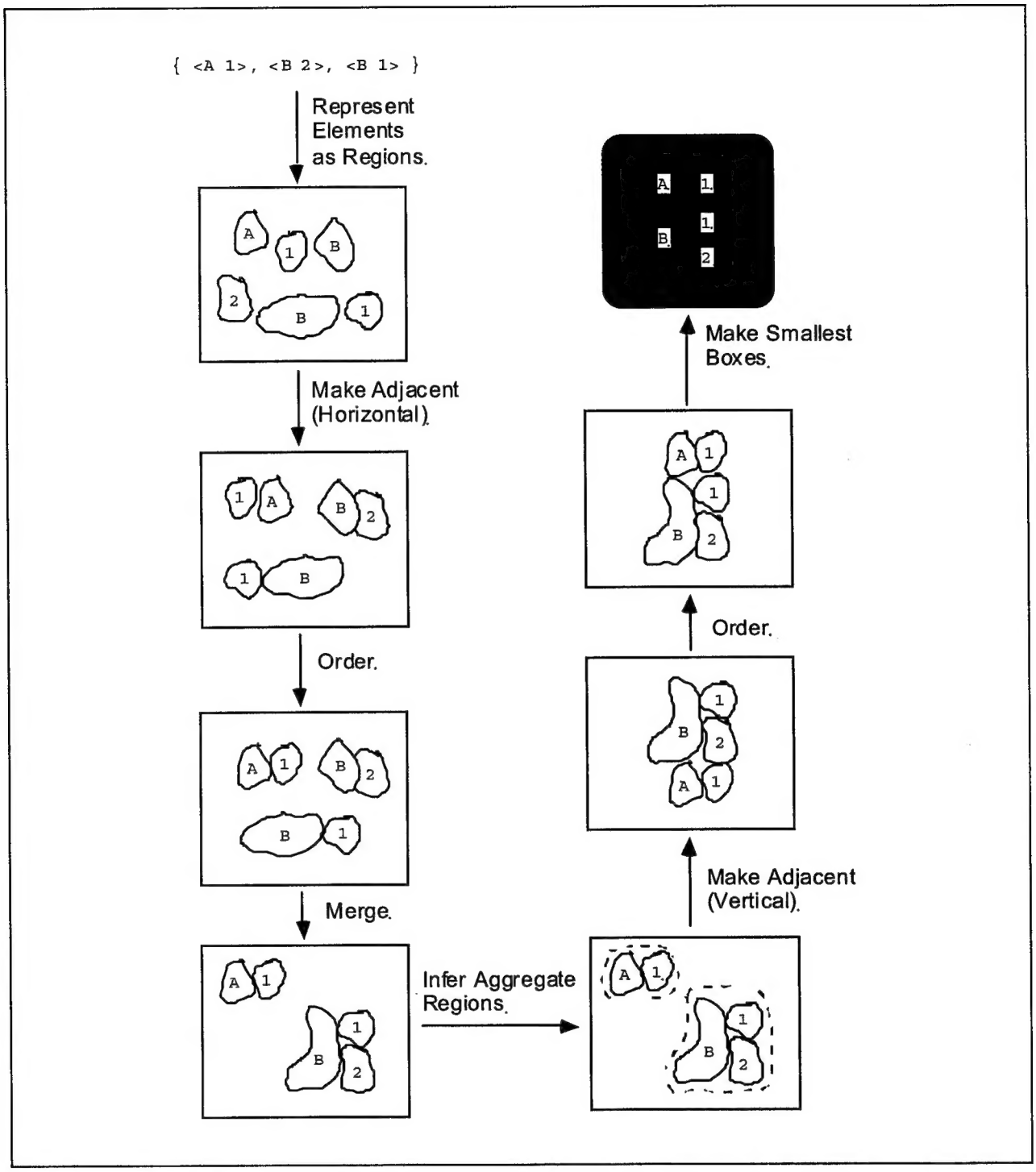


Figure 4. Refinement Example

In addition, Kestrel has technology to synthesize highly-efficient constraint solving algorithms for the domain of scheduling.[Smith93] We believe that our techniques will carry over to allow us to synthesize similar algorithms for constraint solving in two or three dimensions (scheduling is an instance of constraint solving in one dimension, i.e. time), which is ideal for rendering in two or three dimensions. The high speed of the

synthesized constraint solvers is a result of "compiling" some or all of the constraints into the solver, so that the result is a very efficient algorithm specialized for the problem at hand. We believe this holds promise for automating the constraint solving aspects of the visualization process.

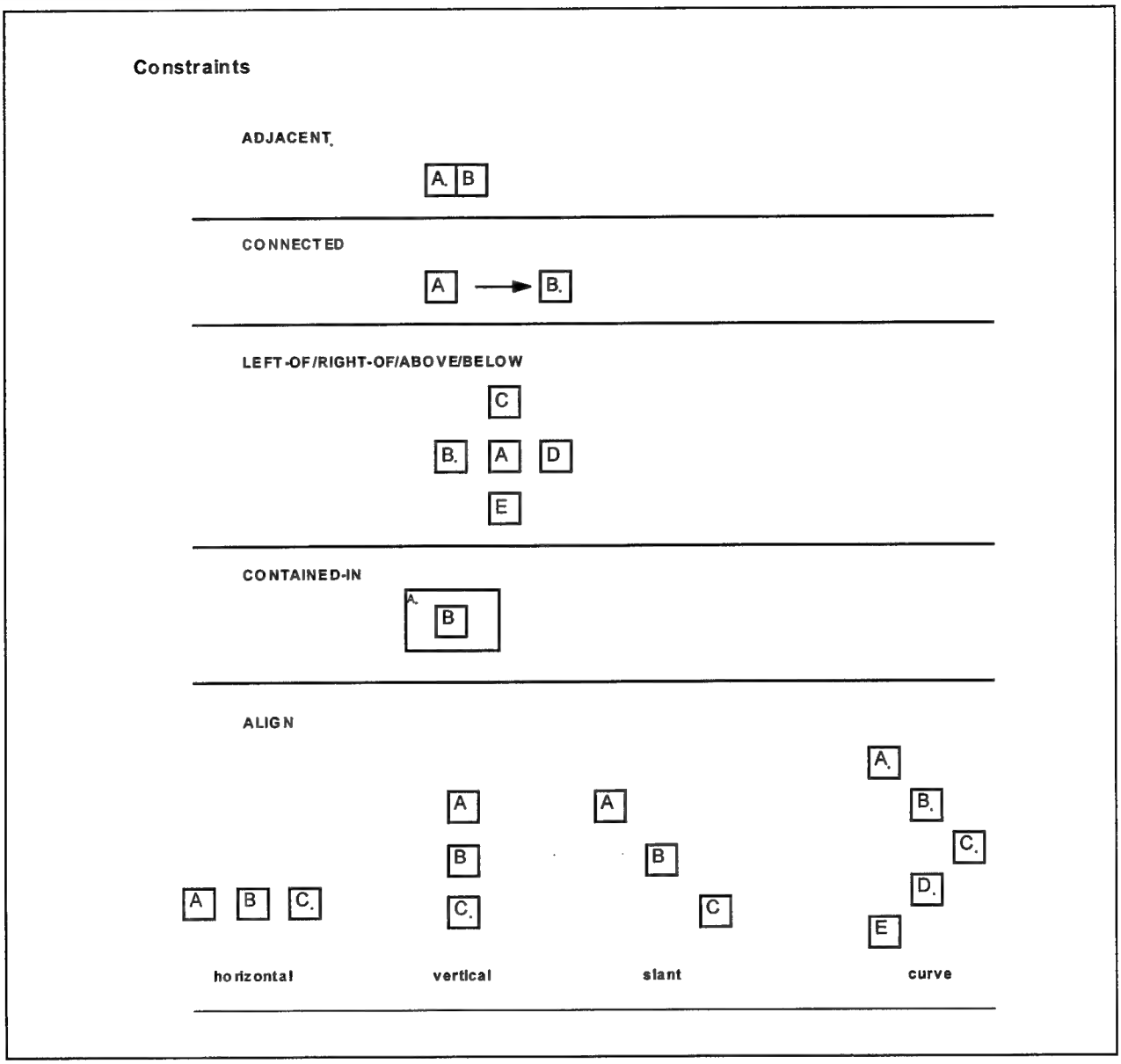


Figure 5. Visualization Constraint Examples

4. Using Automated Visualization Interactively

Thus far we have discussed VIZO in the context of a fully automated visualization synthesis engine, which might be part of a larger system (e.g. for software development). We believe automated visualization can and should be used in interactive contexts as

well. The incremental refinement approach taken in VIZO leads to a natural substrate for supporting a human-directed, machine-driven visualization system. By factoring out the visualization design decisions into independent, composable views, the VIZO approach allows for the creation of a visualization system where the human can try out various techniques very rapidly by choosing which view to apply next, while the machine carries out the task of actually applying the view. The following domain-specific visualization scenario illustrates this point.

4.1. Visualization Example

Scenario: a task force command center has been set up to control a rash of fires that have broken out in the hills surrounding a city. The command center receives reports of the various fires over communication links with helicopter and ground crews throughout the area. The reports are varied in the type and amount of information that they give, and they are constantly changing. The commander wants to get an assessment of the overall situation and decides to concentrate on three pieces of information from each report: (1) the fire's position, (2) its size, and (3) its percent containment. Figure 6 shows the process that occurs, with each rounded box corresponding to the intermediate visualizations that are explored in this process.

The initial visualization in (b) is a simple table generated using the three types of data described in (a). In (c), the users are able to see visually where the fires are in relation to each other because the position of the fires are mapped to positions on the screen. This view is refined further in (d), where size of the fire determines the scale of the representation, and in (f), where the fires themselves are changed from a textual representation to a graphical one (a circle), with shading indicating the current containment level. At this point the users determine that a different approach would be more illuminating, so the visualization is rederived starting with (c) and arriving at (e), where the containment percentage is proportional to the scale factor, and size determinant of the shading, but this time with a legend to explain the shading. Still, the visualization could be made better suited to the task at hand, since the fires that are well contained are depicted as larger and the fires that are poorly contained are depicted as smaller. Thus, (g) rectifies the situation by inverting the relationship between scale factor and containment percentage, so that poorly-contained fires, which are deserving of more attention, appear more prominently in the display. Finally, (i) shows a composition of the two views in (g) and (h). Notice that this is not a simple overlay of renderings (in which the street names would overlap the fires), but rather a composition of the two views, from which a final rendering is generated.

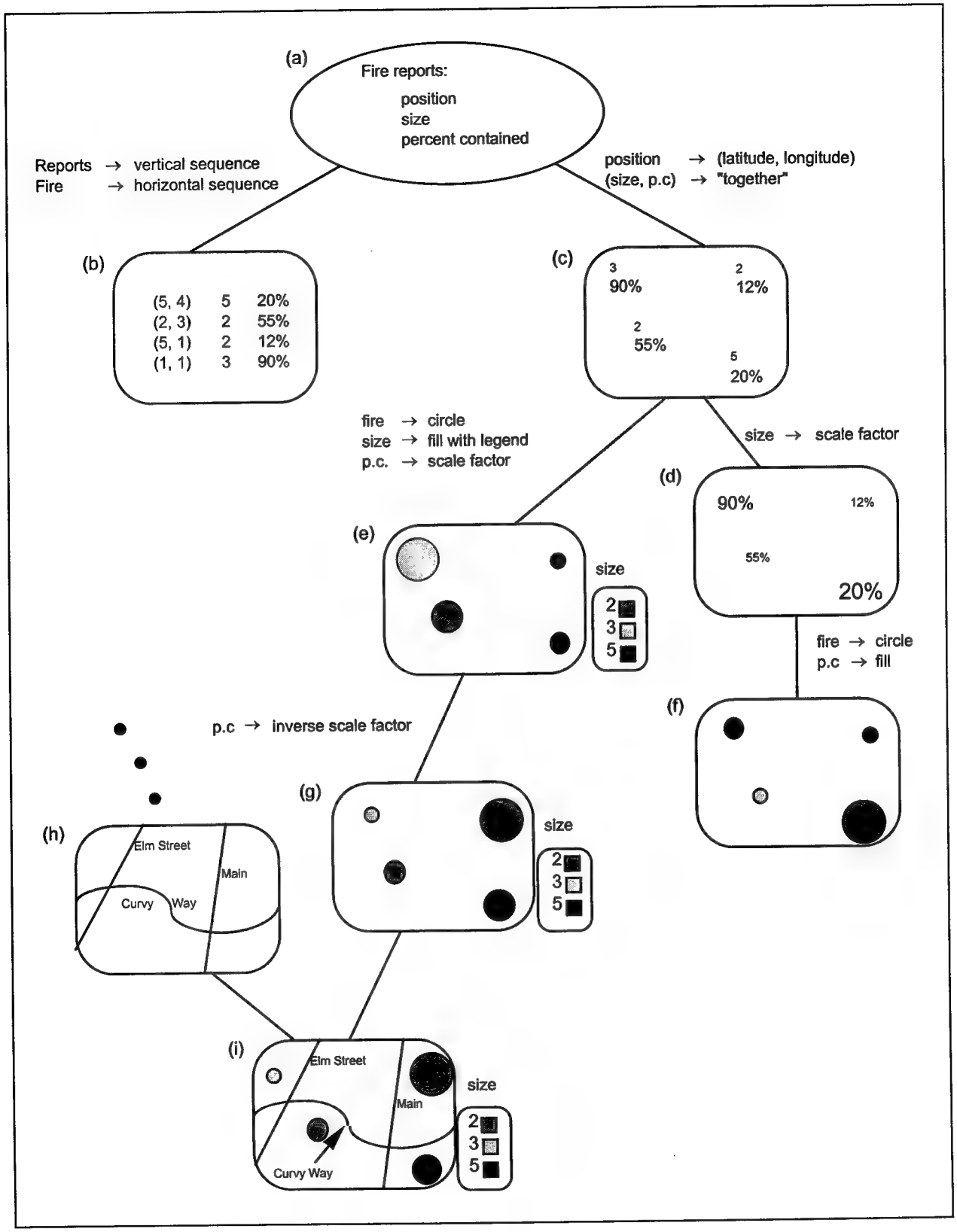


Figure 6. Domain-Specific Visualization Generation Example

4.2. Maintaining Direct Connection Between Data and Visualization

The VIZO approach has an additional advantage in supporting interactive visualization in that the connections between the individual objects on the screen rendering and the data that they represent can be explicitly maintained through the view. This allows for the creation of an interface in which both the data and the rendering can be manipulated by the user, and the other will be updated correspondingly.

5. The VIZO Prototype System: In Detail

The Specware system described in Appendix B is designed to facilitate composition of complex software constructs through the use of fundamental mathematical operations, in particular those from category theory and sheaf theory. We have endeavored to use a consistent design philosophy throughout the implementation of Specware, so that all its components achieve a natural interface to the core system, sharing common abstract structures and methods as much as possible.

For example, the same basic paradigm, and much of the same code, is used when transforming abstract specifications into detailed specifications, when communicating with a theorem prover, and when translating specifications into code. An interesting challenge has been to achieve this same kind of seamless interface for a visualization system.

The approach we've explored starts with a triangle model as depicted in Figure 7, which is an elaboration of that shown in Figure 1. We decompose the structure of a graphical depiction into three main components: the data to be viewed, the view/pattern we seek in the data, and the graphics used to render such patterns. Each of the three components is modeled with a Specware specification, and the two arrows connecting the pattern to data and pattern to graphics are modeled with Specware interpretations.

In all of the examples to follow, the data specification is assumed to be some possibly complex specification describing a scheduling domain with operations on it. The pattern-to-data or "viewing" interpretation projects an image of the pattern specification into a definitional extension of the data specification, i.e., into an extension that merely gives names to some terms already implicitly present in the data specification. It is also correct to think of the view interpretation as locating the pattern in the data.

The pattern specification will vary from example to example, although it is both convenient and reasonable to assume it is constructed from basic set theoretic structures such as numbers, sets, sequences, relations, maps, etc.

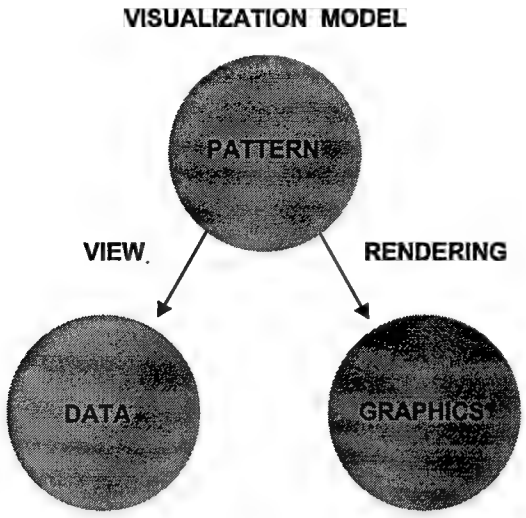


Figure 7

The structure of the graphics specification will depend on the approach chosen, as explained below, but it includes abstract data types for entities such as points, lines, boxes, colors, and dimensions. The pattern-to-graphics, or rendering, interpretation maps the pattern into graphical terms, effectively showing how to depict such a pattern.

At this point, not worrying about the details, we could consider taking the colimit of the diagram shown in Figure 7, combining all of the sorts and operations from both the data specification and the graphics specification. However, that would put all of the significant structure down inside one composite specification, effectively hiding it from observation and manipulation by higher level Specware operations.

A better approach is to notice that if specifications are viewed as topoi, which are a kind of category, then interpretations become functors, which are a kind of structure preserving map from one category to another, which means that given a diagram as in Figure 7, we can construct natural transformations, which are a kind of morphing of one functor into another (or alternatively, a kind of homomorphism between specified portions of specifications). This more elaborated view is shown in Figure 8.

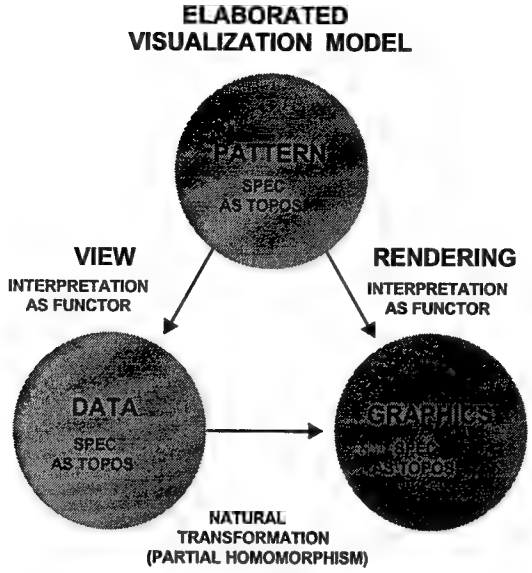


Figure 8

Figure 9 shows a simple example to make this more clear. The pattern specification contains an abstract operation called NatValue that maps Thing to Nat, and the pattern-to-data interpretation maps this NatValue operation onto a Duration operation defined on schedules. The rendering, or pattern-to-graphics, leg of the triangle could also be modeled entirely within Specware as a simple interpretation. In that case the rendering interpretation would map NatValue into something like ColorValue (BarValue, TextValue, etc.), defined in the graphics specification, mapping Thing to Color (Bar, Text, etc.), as shown in Figure 9. The natural transformation then consists of an isomorphism mapping Duration onto ColorValue, Schedule onto Thing, Nat onto Color, etc.

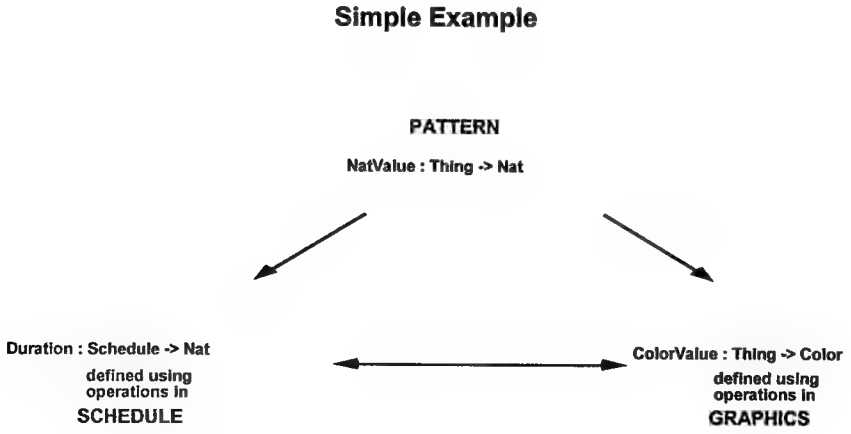


Figure 9

Unfortunately, an undesirable side effect of this approach is that the rules for constructing specification morphisms (which require specification morphisms to be total functions) would require the target Color sort to include the structure of the natural numbers, e.g. to have ColorZero, ColorPlus, etc. In general, that probably requires too much structure on the target sorts, but with the approach described so far, the only way to eliminate some of that structure would have been to simplify the pattern, and that works contrary to desire for the view and rendering interpretations to be orthogonal.

A better approach is shown in Figure 10, where we have connected multiple view/rendition triangles to effectively break the depiction into steps. In the first step, we use the unrestricted pattern specification, and the rendering interpretation maps Nat into an abstract graphics numeric sort that subsequent depictions will specify as a bar, color, sequence of digits, etc. Note that both natural transformations describe isomorphisms, but the second (on the right) is an isomorphism on less structure than the one on the left, since it merely preserves the structure of a total order, whereas the natural transformation on the left preserves the more detailed structure of the natural numbers.

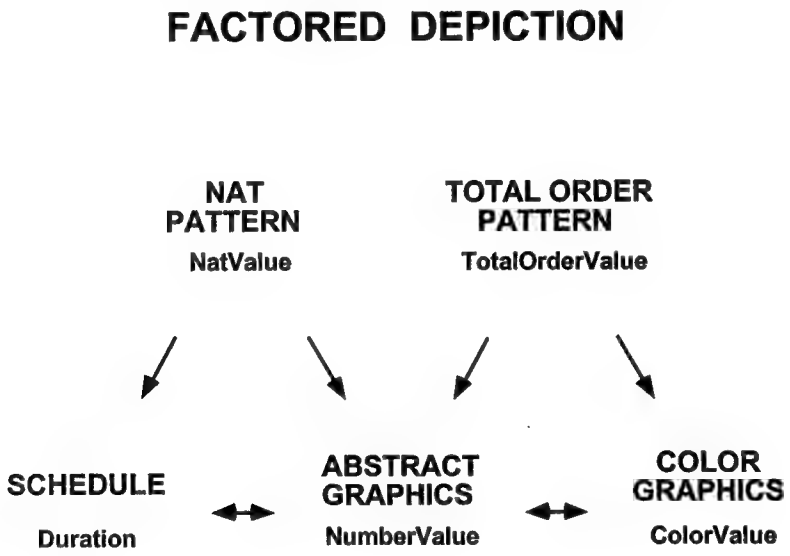


Figure 10

Since the triangles can be composed (side by side) and an identity triangle is easy to construct, we can view these triangles as arrows in a depiction category, as shown in Figure 11. The abstract structure of an arrow in this category is quite general, essentially identifying a homomorphism from part of one specification into part of another, so it seems like a good candidate to add to the general toolkit available in Specware, independent of its use for visualization. This is a good sign and addresses the abstract

concern expressed above for an elegant integration with the core Specware system. In particular, note that accurate and useful rendering of data involves a lot of structure preservation, and we've managed to keep most or all of that regular mapping out at the inter-specification level where generic Specware tools can manipulate it (as opposed to burying it within a single specification in some monolithic presentation). This accords well with our goal of integrating graphical operations with fundamental Specware structures.

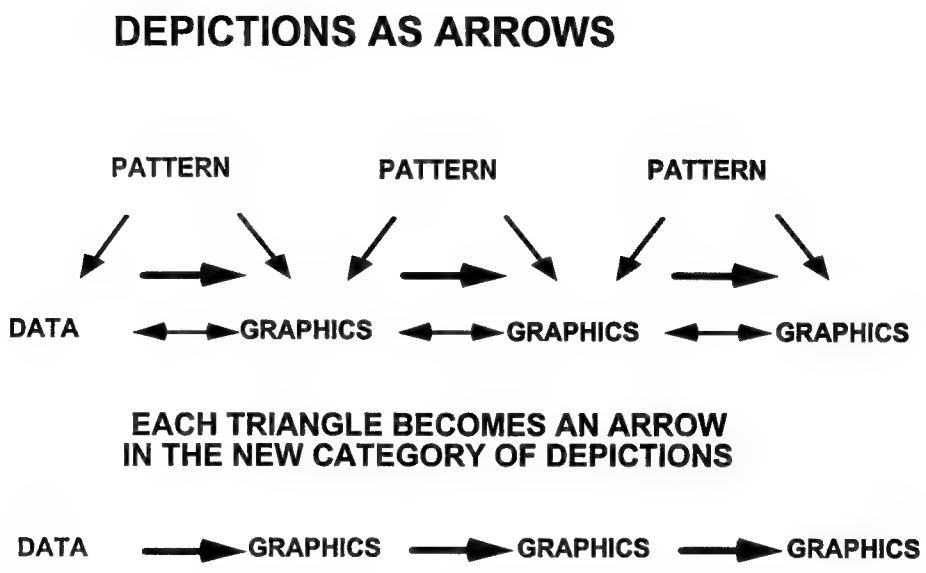


Figure 11

This division of depiction into concatenated steps provides additional benefits. Conceptually, it allows us to factor the depiction into one phase that casts the structure under consideration into abstract graphical terms, and subsequent phases that project those abstract terms into physically realizable ones.

Figure 12 shows this division for a slightly more complicated example in which the pattern is a sequence of intervals. The abstract graphical specification `Abstract_Dimensions` introduces the notion of an abstract dimension, which has sufficient structure to be a good target for intervals, sets, sequences, tuples, relations, maps, and many other set theoretic structures. The first depiction arrow effectively maps the desired pattern into a high (in this simple case, two) dimensional space, then subsequent arrows project that space down into something renderable in real dimensions for space, time, color, etc.

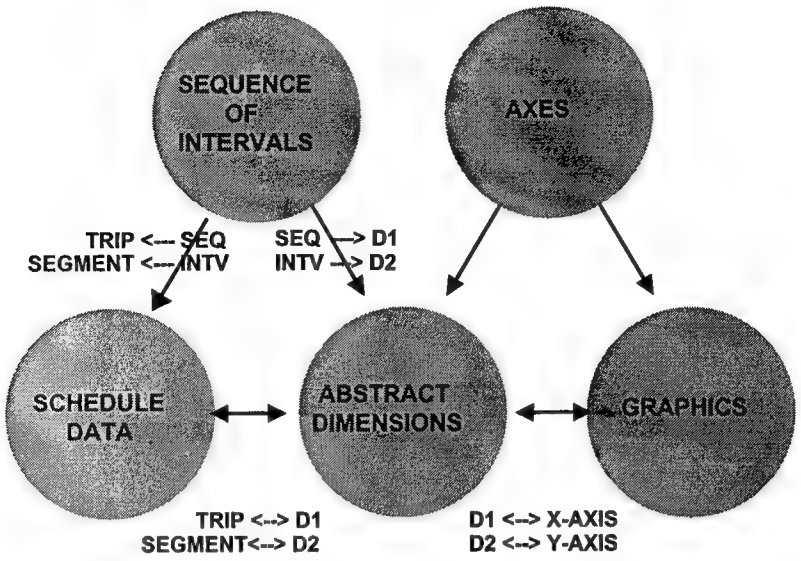


Figure 12

Using an abstract dimensional space as the first depiction target has good computational and user interface properties. It allows us to automate the subsequent depictions with a graphical engine that can produce the effect of what could be done with standard Specware tools, but in an optimized manner that avoids the overhead of reifying as much structure, and under a specialized user interface that allows the user to project dimensions interactively through menus without worrying about the details of constructing actual depiction arrows.

To show how this technology could be developed synergistically with the scheduling and architecture projects underway at Kestrel, a simple project demo involving scheduling data was constructed and given following the 1996 Knowledge-Based Software Engineering (KBSE) conference. The effect was to show in principle how this model could be used to construct a Gantt chart from scheduling data.

To understand how a realistic picture such as a Gantt chart might be constructed, we consider a somewhat more elaborate example. Now the data are a schedule, as in Figure 7, but the pattern, whose top level structure is shown in Figure 13, is a tuple of a string (the schedule-name) and a set (of scheduled assets) whose elements are tuples of strings (asset names) with a sequence (of trips) whose elements are tuples of a string (a trip description) with a sequence (of trip segments) whose elements are tuples of a element from a finite set (loading, flying, etc.) and an interval in time. (See Appendix B for details concerning the specification language *Slang* used in Figures 13 & 14.)

```
specification Pattern-for-Schedule is colimit of
 diagram
    nodes Labeled-Interval,
              Triv1 : Triv,  Labeled-Set-1 : Labeled-Set,
              Triv2 : Triv,  Labeled-Set-2 : Labeled-Set,
              Triv3 : Triv,  Labeled-Set-3 : Labeled-Set
     arcs  Triv1 -> Labeled-Interval  : {E -> Labeled-Interval},
              Triv1 -> Labeled-Set-1     : {E -> Set-Elt},
              Triv2 -> Labeled-Set-1     : {E -> Labeled-Set},
              Triv2 -> Labeled-Set-2     : {E -> Set-Elt},
              Triv3 -> Labeled-Set-2     : {E -> Labeled-Set},
              Triv3 -> Labeled-Set-3     : {E -> Set-Elt}
 end-diagram
```

Figure 13

The first depiction triangle was instantiated with a pair of interpretations, each of which was constructed using standard Specware tools from smaller component interpretations similar to the two shown in Figure 14.

```
Interpretation View-Trip-Segment-As-Labeled-Interval
   : Labeled-Interval => Trip-Segment  is
   mediator  Trip-Segment
   domain-to-mediator      {Labeled-Interval -> Trip-Segment}
   codomain-to-mediator    identity-morphism

Interpretation View-Trip-As-Labeled-Set-of-Labeled-Intervals
   : Labeled-Set => Trip  is
   mediator  Trip
   domain-to-mediator      {Labeled-Set -> Trip}
   codomain-to-mediator    identity-morphism
```

Figure 14

The primary pattern-to-graphics interpretation mapped tuples, sequences, strings, etc. in the pattern onto various kinds of virtual dimensions in a graphics specification. Then a prototype version of the user-interface code was used to dynamically assign virtual dimensions to real dimensions, as explained next.

For this example, we generated 18 such dimensions, as follows:

```
D1        :  schedule              :  two element tuple of <D2, D5>
D2        :  schedule name         :  sequence over D3 x D4
D3, D4    :  character region      :  two orthogonal finite dimensions
D5        :  scheduled assets      :  sequence over D6
D6        :  scheduled asset       :  two element tuple of <D7, D10>
D7        :  asset name            :  sequence over D8 x D9
D8, D9    :  character region      :  two orthogonal finite dimensions
D10       :  trips                 :  sequence over D11
D11       :  trip                  :  two element tuple of <D12, D15>
```

```
D12        :  trip description     :  sequence over D13 x D14
D13, D14   :  character region     :  two orthogonal finite dimensions
D15        :  trip segments        :  sequence over D16
D16        :  trip segment         :  two element tuple of <D17, D18>
D17        :  segment status       :  finite set {loading, flying, unloading}
D18        :  segment duration     :  interval in time
```

There are properties on individual virtual dimensions, and constraints among the virtual dimensions. For example, the virtual dimensions used for sequences must be mapped onto real dimensions with extent, e.g., space or time but not color, whereas dimensions that represent a single-valued attribute such as D17 (segment status) are allowed to map onto almost any dimension: space, time, color, a set of patterns, etc. Since dimensions D3 and D4 form a character plane, they must map to a pair of real dimensions that are orthogonal to each other, but it is allowed (even preferable) for both D3 and D18 to map to the same real X dimension.

The rendering engine interacts with the user to map a set of virtual dimensions into real dimensions, after which the data are automatically displayed according to that format. The user can then change this mapping indefinitely, occasionally asking for the data to be redisplayed in whatever format is current.

Because the number of real dimensions is quite limited, we need to use rules, heuristics, and user guidance to consistently and effectively map the myriad virtual dimensions down to the available real dimensions. There are several strategies we can employ. One is to increase the number of available real dimensions, a second is to hyperlink, a third is to collapse redundant dimensions, a fourth is to tile one virtual dimension within another, and a fifth is to allow the option of simply ignoring some dimensions. These five approaches are described in the following paragraphs. There could be other such strategies--this is not meant to be an exhaustive list.

Adding more real dimensions is not as difficult as it sounds. Since many of the virtual dimensions we need to map represent small finite sets, even small sets of visually distinct attributes can be exploited to add new dimensions. For example, in addition to X, Y, and Z spatial dimensions, we can include color dimensions such as hue or intensity, or dimensions that consist of a set of distinguished patterns. In the example above, we could map dimension D17 [loading, flying, unloading] onto colors [blue, red, green] or onto patterns [thin bar, thick bar, thin bar], etc.

Hyperlinking allows us to use an alternative part of the screen to actively show additional information pertaining to the location specified by a pointer. For example, dimensions D12, D13, D14 could be mapped into an active window such that the relevant string would be depicted whenever the mouse cursor was over a (real) X,Y coordinate used as part of the depiction of some trip.

Some dimensions can be collapsed into others. In the example above, for any given asset, differing coordinates in D15 (sequence of segments) imply differing coordinates in D18

(time), so we can collapse D15 onto D18 without losing any information, as indicated in Figure 14. Collapsing D15 onto D18 assumes we map D17 (segment status) to color or some other dimension distinct from D18 otherwise the association between trip segments and trip status may be lost. Likewise, we might be able to collapse D10 (sequence of trips) into D18 as well, assuming we map D12 (trip description) to a hyperlink dimension or some other dimension distinct from D18.

Tiling can be useful when one or more dimensions are finite. For example, the dimensions used to draw an individual character are finite, hence if the (finite) virtual X dimension for characters in some sequence is mapped into the real X dimension, a sequence of such characters can be depicted by tiling that finite dimension into repetitions along the X axis, using coordinates in the sequence dimension to select particular tiles. For another example, the durations in dimension D18 are bounded, and in particular are bounded from below by 0, so if we map D18 onto the real X axis we can map some other virtual dimension that is finite (e.g. D17, D12, or even D5) into the unused part of the real X axis. Sequences and tuples in general often allow these two different kinds of tiling, respectively.

Ignoring a dimension is useful when we just don't care to see some particular information, or when the dimensional clutter is otherwise too great and we need to look at lower-dimensional slices. In this example, we might choose to ignore dimension D12 (trip description) and possibly D7 (asset name) if we were just looking to see if a feasible schedule exists. This can be viewed as a degenerate kind of hyperlinking to oblivion.

For the demo given in September of 1996, one result of interactively making choices as described above was the screen partially shown in Figure 15. Other formats, such as a swapping of the X and Y axis, were possible but were more crudely rendered, since the intent of that demo was limited primarily to showing that a traditional Gantt chart could be generated. In this picture, a trip segment status has been mapped to color using the rules {loading -> blue, flying -> red, unloading -> green}, time has been mapped on the X axis, the set of assets have been arranged along the Y axis, and labels for both the entire schedule and each asset have been oriented as normal horizontal text. Some dimensions were ignored to make the rendering compact.

A mature graphics engine would probably provide a more graceful facility than a simple menu for describing and assigning virtual dimensions to actual (renderable) dimentsions. It would also contain a constraint system to ensure that user choices for embedding dimensions are consistent, to automatically make choices based on constraint propagation, and to allow default embeddings based on heuristics and/or user profiles, etc.

Figure 15

Overall, the model we have developed shows promise as an extremely flexible tool for rendering complex data. By making rendering decisions about the various dimensions as orthogonal as the data permit, we expect to be able to encourage depiction experiments that otherwise might not occur to people, and which would be too expensive to explore with traditional programming techniques. That kind of high level experimentation provides an excellent medium for achieving optimal (and occasionally breakthrough or paradigm-shifting) results in real world applications.

6. The Role of Sound

Sound is an important, but under-utilized and under-explored channel for software understanding. Note the following key features of sound for understanding:

- Sound provides an **extra channel** for providing information to the programmer, which can be used in parallel with visualization.
- It can provide **new ways** of understanding programs. For example, sounds gives a better sense of timing than visual displays.

With the help of a consultant, Stanley Jordan, we explored the possible uses of music and sound to aid in the development and understanding of software.

The following are the areas where we considered using sound:

- During Editing, for keeping track of
  - (a) what you are doing, and
  - (b) the state of program.
- During Debugging
- Understanding and managing the run-time behavior of the program, e.g.
  - (a) tuning performance, and
  - (b) monitoring progress.
- User interface enhancement
- Program/Derivation Understanding

As a relatively simple experiment we took the output of our scheduling program and converted the trips into MIDI data. We used the natural mappings of time in the schedule to musical time, and quantity to volume. Other attributes such as kind of cargo were mapped to different instruments of pitches. We tried different assignments so as to emphasize distinctions between different attributes. Playing the schedule at an accelerated rate gave an interesting overview of the schedule. We view this work as promising, but still quite experimental.

As an avenue for future research, we note the parallel between transformational synthesis (as in KIDS/Specware derivations) and certain methods of music composition/analysis (e.g. Schenkerian). A possible connection to exploration involves finding musical refinements corresponding to different program refinements. Given such an approach, one could derive a piece of music in parallel to deriving a program. The structure of the music would have certain correspondences to the program. By analogy, the music for an inefficient program might have excessive repetition. Extending the analogy further, by introducing a variable for a repeated expression might have a musical correspondence such as using only the first few notes of a long phrase instead of repeatedly using the whole phrase.

A general problem with using automatically generated music is ensuring that it sounds reasonable. Inherent in the approach of the previous paragraph is the necessity of formalizing musical knowledge in a way comparable to our formalization of programming knowledge. This musical model would have to be rich enough to ensure results that meet standards of musicality. One possibility is to create models beginning with pre-composed pieces (by a musician), which are then parameterized for use in derivations.

In brief, we believe that sound will play an important role in program understanding in the future, as the complexity of software systems being built increases each year. Our initial findings in this area indicate that graphics are well-suited for uncovering static structure, whereas sound may help us uncover dynamic structure or timing.

7. Summary

This research explored various ways of formally visualizing software systems. We explored relationships between views, data, and renderings in a formal setting where visualizations were created by mapping data onto abstract views that were refined into renderable entities. We explored the feasibility of our approach by developing both a formal specification and an implementation of a software system for visualizing scheduling information. We also briefly explored the role of sound as a visualization aid, and found that although more work remains to be done, sound can be effectively used to analyze structure. Our feasibility demonstration conducted in September 1996 proved we could visualize systems using formal techniques and that such visualizations could be co-derived with system software. However, much work remains. For example:

- We feel that better constraint-solving technology – one handling more complex and aggregate constraints -- will greatly aid the system, and furthermore, such technology is likely to be built within the next few years as the importance of constraint-based reasoning is being realized in the larger community.
- A better characterization of the constraints associated with adding and using real (renderable) dimensions is needed. For example, under what circumstances can a dimension be collapsed onto another? How can these constraints be formalized and used in the VIZO system?
- How can we better support the task of assigning virtual dimensions to real dimensions? What role should heuristics play and how can they be formalized for use within VIZO?
- Finally, the role of sound in analyzing structure needs to be better understood.

In brief, we feel that we have demonstrated the validity of our premise, that visualizations can be co-derived with software and used to aid system understanding.

Bibliography

[GRW96]
Gerken, Mark J., and Roberts, Nancy A. and White, Douglas A. "The Knowledge-Based Software Assistant: A Formal, Object-Oriented Software Development Environment." *Proceedings of the National Aerospace and Electronics Conference 1996 (NAECON '96).* (2):511-518. Dayton, OH: IEEE Press number 96CH35934, 1996.

[GLB+86]
Cordell Green, David Luckham, Robert Balzer, Thomas Cheatham, and Charles Rich. "Report on the Knowledge-Based Software Assistant." *Readings in Artificial Intelligence and Software Engineering,* edited by Charles Rich and Richard Waters, 377-428, Morgan Kaufmann Publishers, Inc., 1986.

[JS93]
Jullig, Richard and Srinivas, Y.V. "Diagrams for Software Synthesis." *The Eighth Knowledge-Based Software Engineering Conference*, 10-19. Los Alamitos CA: IEEE Computer Society Press, September 1993.

[JS95]
Jullig, Richard and Srinivas, Y. V. *Specware: An Advanced Software Development Environment.* Technical Report RL-TR-95-14, Rome Laboratory, Rome NY 13441-4505, 1995.

[Smith93]
Smith, Douglas R. "Transformational Approach to Transportation Scheduling." *KBSE '93: The Eighth Knowledge Based Software Engineering Conference.* 60-68. IEEE Computer Society Press, 1993.

Appendix A. Visualization in the Specware Framework

A.1. Overview

This appendix describes an implementation of visualization refinement in Specware as part of the VIZO prototype. This appendix is organized as follows:

- Section A.2 contains a worked example;
- Section A.3 describes diagram interpretation;
- Section A.4 describes multiple representations; and
- Section A.5 contains the Specware specifications referenced in the preceding sections.

A.2. Worked Example

This section describes a key part of the VIZO prototype: a mechanism for hooking the visualization domain into the refinement process of Specware. We explored this mechanism by creating and manipulating specifications for graphical user interfaces (GUIs).

A.2.1. GUI Template

The specification **`GUI Template`** (given in Section A.5) is a template for constructing Specware *interpretations* which map abstract views to concrete visual representations. There may be more than one concrete object for each abstract object, and each concrete object may use a different representation scheme. Intervening between the concrete and abstract objects are representation objects. This very general scheme allows us to not only manipulate screen objects by modifying the underlying data or entity being represented, but it also allows us to manipulate the data via, say, mouse operations on the screen objects.

Figures A.1 through A.3 show three different aspects of the GIU Template. Figure A.1 shows GUI Template at an overview level. Figure A.2 shows the internal structure in more detail, and Figure A.3 illustrates the relationship between the pieces of the internal structure and how they enable flexible visualizations. Figure A.4 shows how the GUI Template specification is an interpretation scheme called Triv-to-Dobj, whose pieces can later be refined to form interpretations for specific visualizations. (See Section A.3.2.)

GUI Template: Abstract as Concrete

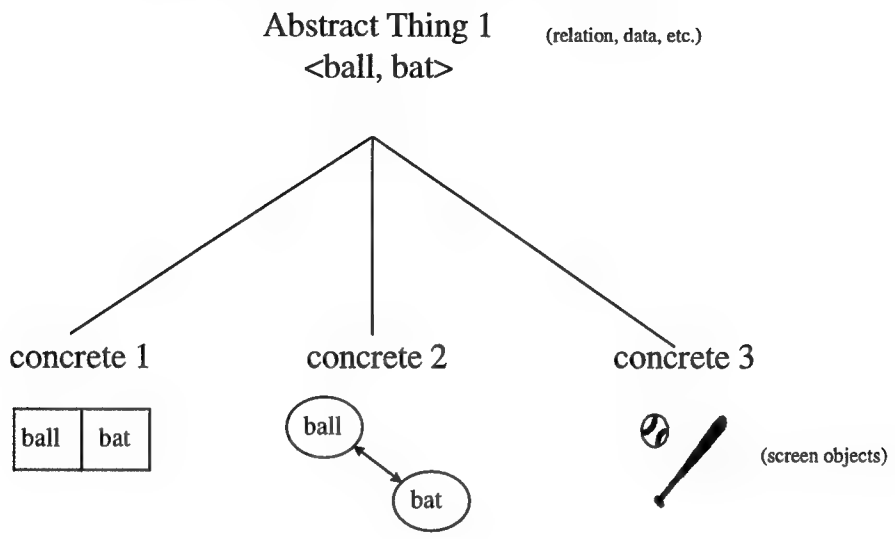


Figure A.1

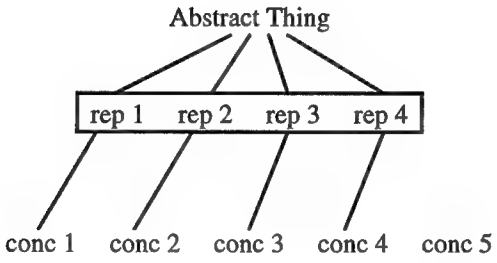


Figure A.2

In Figure A.2, rep 1, rep 2, rep 3, and rep 4 all represent the same abstract entity. Similarly, conc 1, conc 2, conc 3, and conc 4 are concrete (displayable) representations of the abstract entity. Note that conc 5 is a concrete representation that is not a valid representation of Abstract Thing.

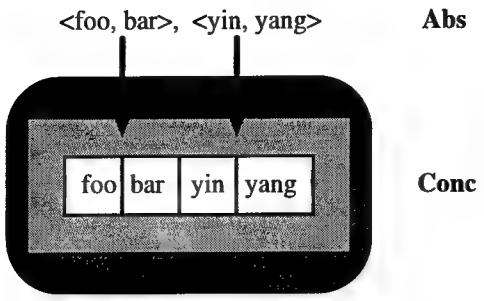


Figure A.3

Consider the example depicted in Figure A.3 where two pairs of symbols are represented on the screen as two pairs of adjacent boxes with text labels. "Pairness" is being represented using adjacency. It happens that "bar" and "yin" have been placed adjacent to each other, however, the pair <bar, yin> is not in the set of pairs being represented. Thus, below we see that the three adjacent pairs of labels are in the concrete sort, but only two of the three are in the rep sort. Rep is a *subsort* of conc. Notice that in this example, unlike the diagram above, each abstract element has only one rep.

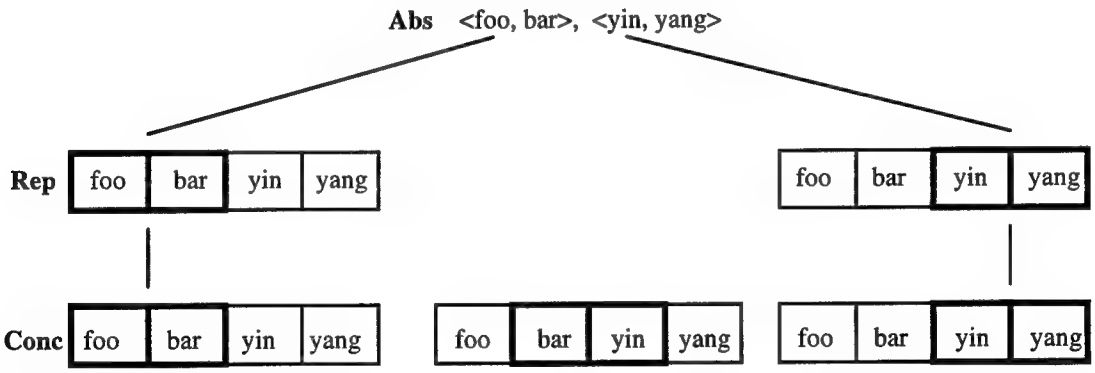


Figure A.4

A.2.2. Display Objects

Figure A.5 shows the root of a hierarchy of display objects, and includes a legend for Figures A.5-A.8. Notice that the named objects are names of specifications which appear in Section A.5. Figure A.6 is an annotated version of a Specware diagram for the interpretation Pair-to-AdjXPair, in which the GUI Template described in Section A.2.1 is instantiated. As it's name suggests, Pair-to-AdjXPair refines an abstract pair of items into a pair of visual representations (as regions) of those items, with the added constraint that

their regions must be horizontally adjacent. Figure A.7 shows a three-dimensional Specware diagram of an interpretation (including its component parts) of a particular pair into a pair of adjacent visual regions. Notice that the diagram in Figure A.6 of Pair-to-AdjXPair is actually a subdiagram of this full diagram (it is the entire backmost plane of the 3D shape). This illustrates one manner in which refinements themselves can be composed and refined in Specware. Finally, Figure A.8 shows the same full example, but abstracted slightly to show how it is constructed incrementally, starting with the GUI Template on the left and ending with the final refinement on the right.

In order to better illustrate the visual language of Specware, we will explain in detail the full diagram in Figure A.7. The diagram is structured as a three dimensional lattice with the lattice points being specifications and the arrows being morphisms between the specifications. Each specification has a textual identifier as well as a graphical depiction. The morphisms do not show their full structure here, however notice that some morphisms are annotated with a small "d" indicating that they are definitional extension morphisms. The lattice is basically a three dimensional box: three specs wide, three specs tall, and two specs deep. There is an additional buttress plane which juts out from the middle front vertical line towards the viewer.

Each plane in the structure highlights a semantically significant relation among the specs which participate in that plane. For instance, the topmost plane consists of abstract data type specs, while the bottom-most plane consists of visual element specs. The intervening plane contains the mediator specs which complete the interpretations (and interpretation schemes) between the abstract and the visual specs. We have used a depiction of a computer screen for each spec at this level, indicating that these are specs which can be used directly to create automated visualizations. The specs in the left and right most vertical planes serve as component specs to the specs in the middle vertical plane. For example, The spec labeled "Person" and the spec labeled "Name" are components that are imported to create the spec labeled "Pair(Person, Name)".

Working from back to front we see an instantiation process. The back-most plane is most generic (e.g. it contains a spec, "Pair", which describes all possible pairs of elements in the universe). The front plane of the box instantiates the specs in the back plane and thus restricts the possible models for each spec (e.g. all pairs of persons and names). The buttress instantiates the specs further by choosing one particular instance (e.g. the pair whose first element is ss#123-45-6789 and whose second element is John Doe).

In choosing representations for abstract concepts, especially when several similar concepts are simultaneously being displayed, we have found it useful to take a "minimal model" approach. The idea is to represent the concept as abstractly as possible (so that it covers all intended models) while still conveying the concept unambiguously. Also important is to not introduce spurious relationships.

A.3. Further Refinement

After our initial model of visualization in Specware, we began to refine it based on our experience and improved theoretical understanding.

A.3.1. Multiple Representations

We begin with a theory of multiple representations (see Figure A.9). A concrete representation of an abstract object A is an object C with an abstraction map *abs* : C --> A and a representation map *rep* : A --> C. We require that *abs* ° *rep* = id, so that *rep* is injective and *abs* is surjective. Thus, a representation may have *more* information than the abstract object that it represents. For example, a representation of a person's name may have a position on the screen and a bitmap displaying the text in some font. However, since rep is injective, the concrete object always contains the full content of the abstract object.

Given two concrete representations (C_1, rep_1, abs_1) and (C_2, rep_2, abs_2) of an abstract object A, the images of A in C_1 and C_2 (via rep_1 and rep_2) are isomorphic via $f_{12} = rep_2 \circ abs_1$ and $f_{21} = rep_1 \circ abs_2$. That is,

$$f_{21} \circ f_{12} \circ rep_1 = rep_1$$
$$f_{12} \circ f_{21} \circ rep_2 = rep_2$$

The entire representations are not isomorphic, just the images of the abstract object within the representations.

This theory appears to have two problems. First, it does not appear to account for representations that show only part of an object, since the representation map must be injective. For example, suppose we want to show only the parity of an integer (whether it is even or odd). Second, the primacy of the abstract object seems to contradict the idea that all representations are on equal footing.

By viewing the same theory in a slightly different way, we can remove both difficulties. In place of an abstract object, we choose *view*, some aspect of the object we would like to present. For example, a view for parity would be {0,1}. In the above discussion, the view replaces the abstract object, and the abstract object becomes just another concrete object. This method also allows multiple views of the same object (see Figure A.9).

The theory of multiple representations, as presented here, is formally identical to the theory of fiber bundles, which is dual to patching theory. In patching theory, we are concerned with compatible covers of a large object, while in bundle theory we are concerned with compatible representations (see Figure A.10).

Finally, we see that this theory also applies to classes of objects with internal structure, such as lists. In this case, the abstraction and representation functions are list homomorphisms:

```
(rep (nil1))       = (nil2)
(rep (cons1 a l)) = (cons2 a (rep l))
```

A.3.2. Specifications

To build these ideas into a specification for pairs of people and names, we need several components:

- A theory of pair homomorphisms
- A theory of multiple representations
- A theory of visualizations of pairs

These theories are shown in Section A.5.2. The theories of pair homomorphisms and multiple representations are straightforward. We then glue three copies of the representation theory to two copies of the pair homomorphism theory to capture the notion that both the abstraction and representation maps are pair homomorphisms.

To visualize pairs, we need the idea of a pair as a displayable object, whose components are displayable objects, and whose components are adjacent. We build this theory from components using a ladder construction. We start with OBJECT, the basic theory of objects. We specialize OBJECT to make PAIR and DISPLAY-OBJECT, then use a colimit to generate DISPLAY-PAIR. In this way, we combine two independent theories automatically.

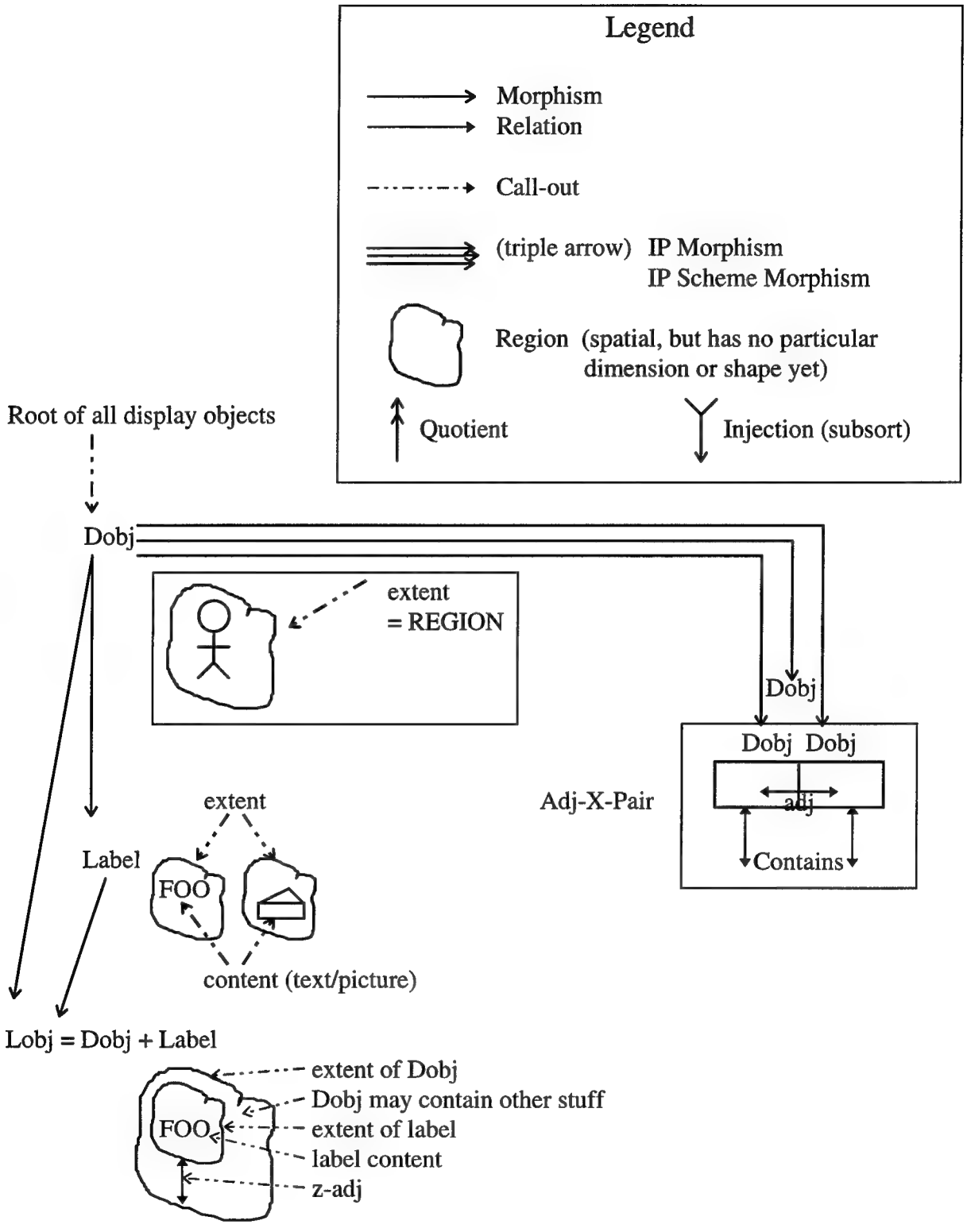
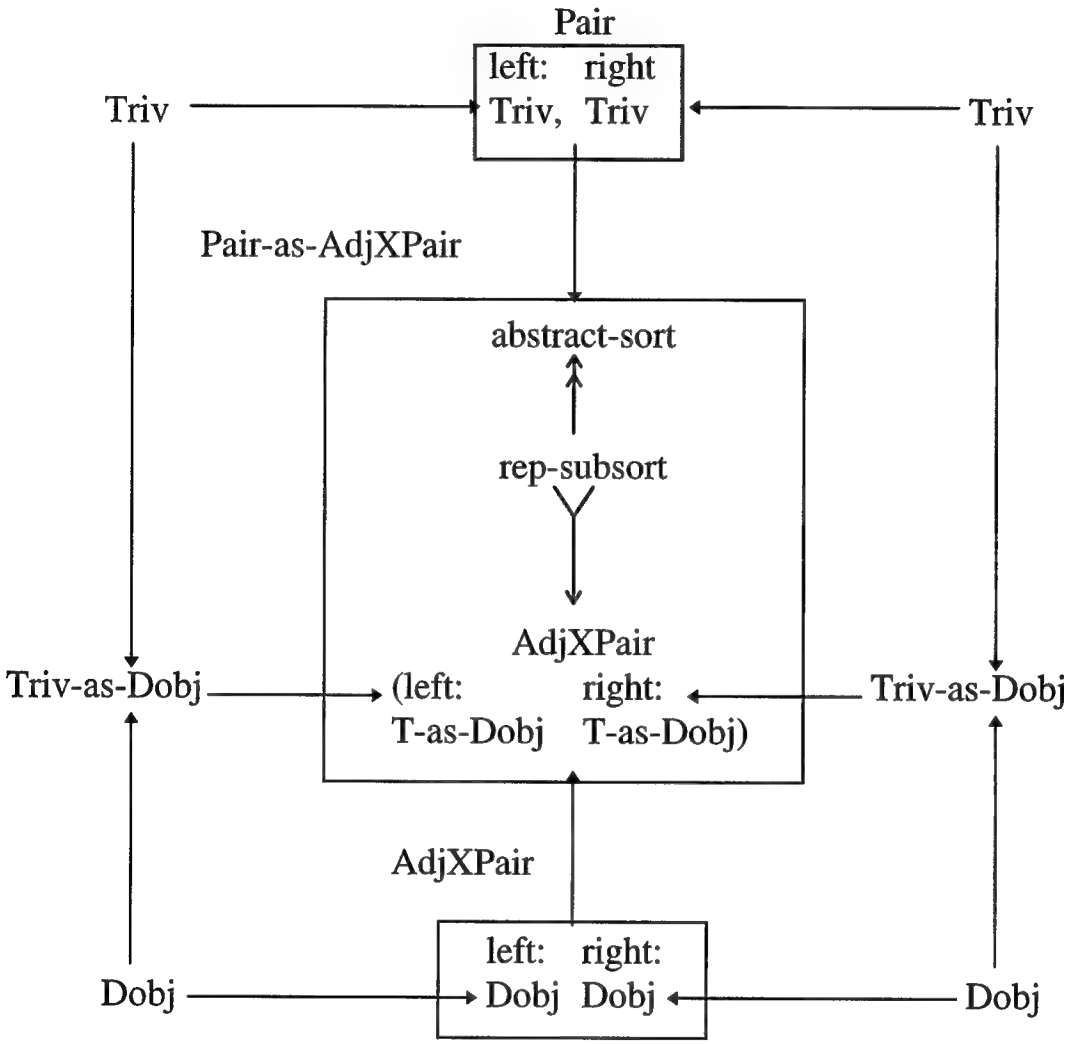


Figure A.5: Display Objects

After specializing PAIR to pairs of display objects and further to pairs of *adjacent* display objects, we use colimits twice more generate *displayable* pairs of adjacent display objects. Colimits solve the multiple inheritance problem by allowing us to control sharing. Thus, we can specify new concepts where they are fundamental and propagate them to where they are needed. For example, "displayability" is a property of objects, not of pairs, but we can propagate it to pairs using a colimit.



Result: Representation of Pair as AdjXPair is an instance of GUI template, and can therefore be reused in further construction.

Figure A.6: Pair to AdjXPair

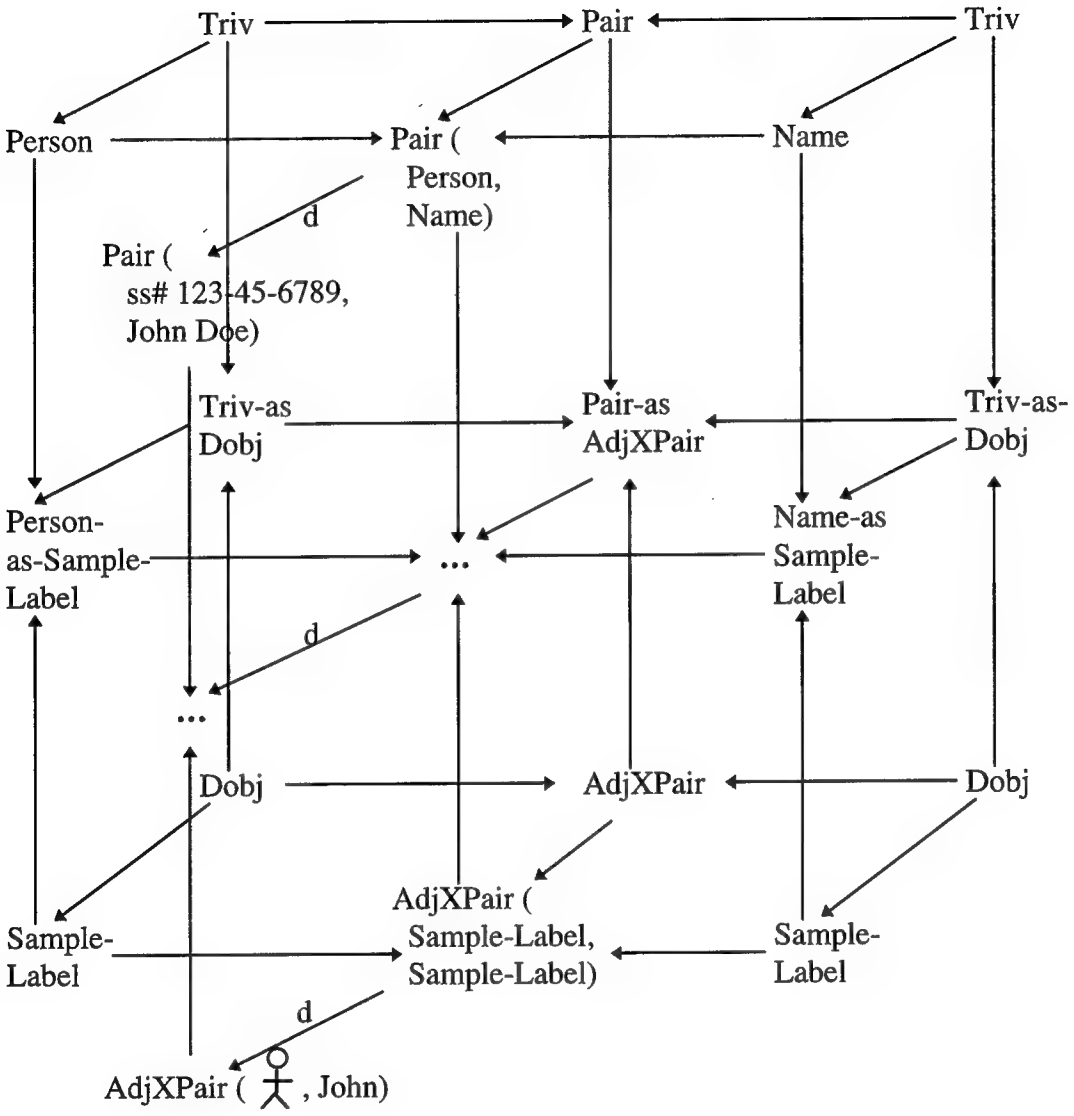


Figure A.7: Full Example Refinement

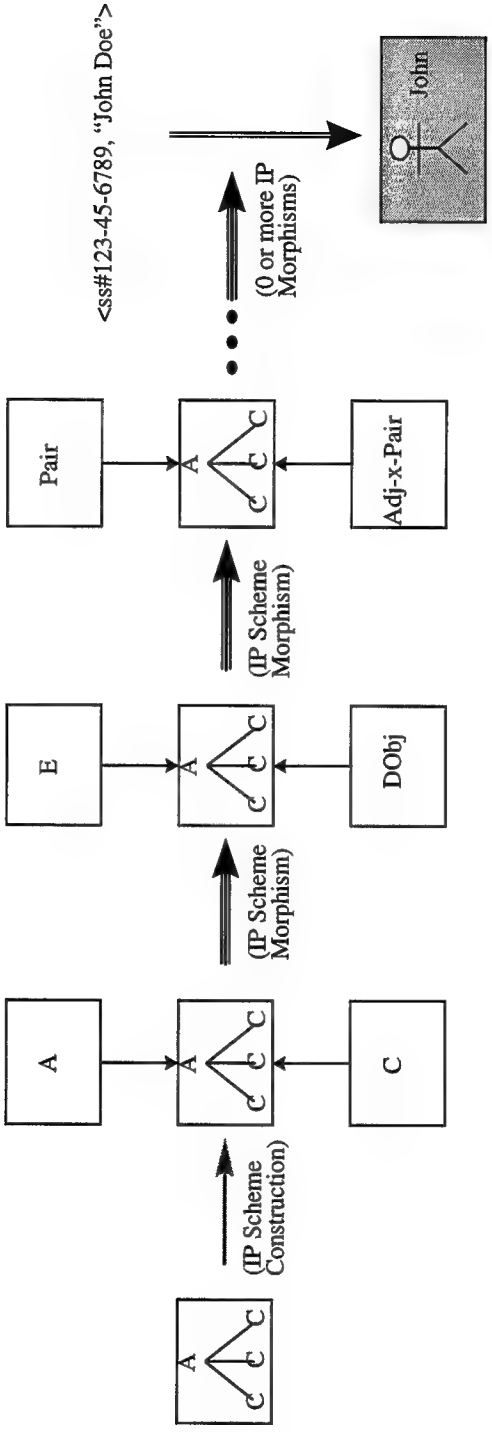


Figure A.8: Visualization Refinement Steps

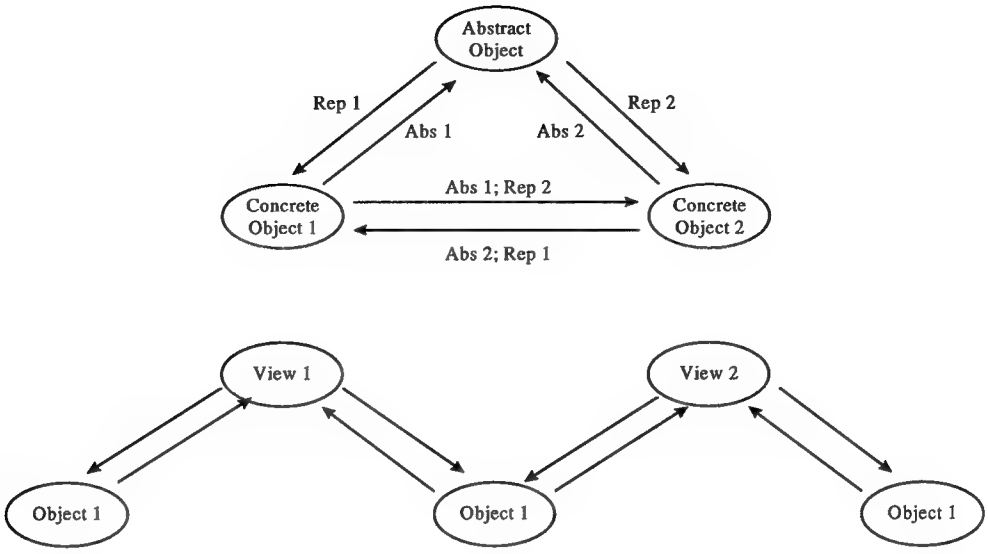


Figure A.9

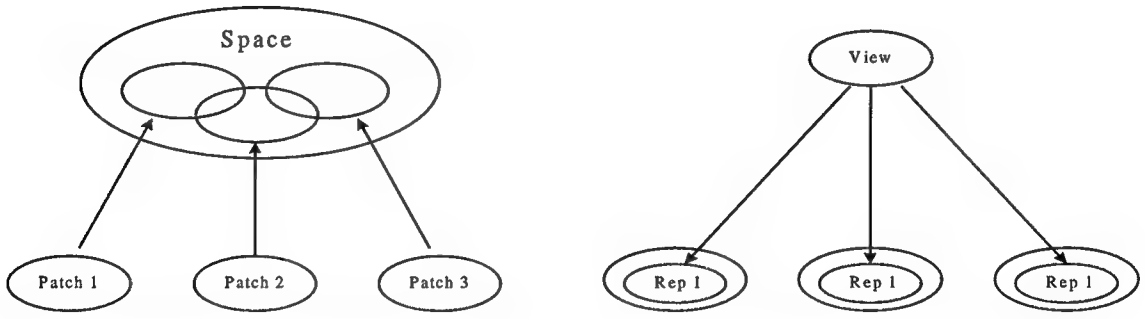


Figure A.10

A.5. Specware Specifications

A.5.1. GUI Template

```
%% -*- Mode: RE; Package: SPEC; Base: 10; Syntax: Refine -*-
!! in-package("SPEC")
!! in-grammar('SLANG::SPEC-GRAMMAR)

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Auxiliary files:
%% relations.re, tuples.re
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Abstract description of regions
%% The model to keep in mind is a 2.5-D display (i.e., 2-D display
%% with layering)

spec REGION is
  sort Region

  %% adjacency in different directions

  op adj-x? : Region, Region -> Boolean
  op adj-y? : Region, Region -> Boolean
  op adj-z? : Region, Region -> Boolean

  %% (contains x y) is to be read as x contains y

  op contains : Region, Region -> Boolean

 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Abstract display objects encompass anything that can be
%% displayed. In an OOP system this would be the root of a hierarchy
%% of displayable objects.

%% For now, the only thing we assume about abstract display objects is
%% that each such object has an underlying region (its extent). This
%% theory could be augmented with other generic attributes (e.g.,
%% position) and generic methods (e.g., mouse-handlers).

spec DISPLAY-OBJECT is
  import REGION

  sort D-Object
  op extent : D-Object -> Region
 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Labels are some display objects which some "content". For now, this
%% content is either a picture or a piece of text (we don't specify
%% what pictures and text are).

spec TEXT is sort Text end-spec
```

```
spec PICTURE is sort Picture end-spec

spec LABEL is
  import translate DISPLAY-OBJECT by {D-Object -> Label},
         TEXT, PICTURE

  op content : Label -> (Picture + Text)
 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Any display object can be labelled by attaching a label to
%% it. "Attaching" here means that an object and its label are
%% adjacent in the z-direction.

spec LABELLED-DISPLAY-OBJECT is
  import
  translate
  colimit of diagram
               nodes DISPLAY-OBJECT, REGION, LABEL
             arcs  REGION -> DISPLAY-OBJECT : {},
                   REGION -> LABEL : {}
           end-diagram
  by {D-Object -> LD-Object}

  op lbl : LD-Object -> Label

  axiom label-is-adj-z is
    (adj-z? (extent ld-obj) (extent (lbl ld-obj)))

 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Pair of objects which are adjacent in the x-direction.

%%  +----------+
%%  |+---+---+|
%%  || L | R ||
%%  |+---+---+|
%%  +----------+

spec ADJ-X-PAIR is
  import
  translate
  colimit of diagram
               nodes pair : DISPLAY-OBJECT,
                     left : DISPLAY-OBJECT,
                   right: DISPLAY-OBJECT,
                   REGION
             arcs  REGION -> pair : import-morphism,
                   REGION -> left : import-morphism,
                  REGION -> right: import-morphism
           end-diagram
  by {pair.D-Object  -> Adj-x-Pair,
      left.D-Object  -> Left-D-Object,
      right.D-Object -> Right-D-Object}

  op pair-left   : Adj-x-pair -> Left-D-Object
  op pair-right  : Adj-x-pair -> Right-D-Object
```

```
   %% left and right regions are contained in the overall region

   axiom (contains (extent p) (extent (pair-left p)))
   axiom (contains (extent p) (extent (pair-right p)))

   %% left and right regions are adjacent in the x-direction

   axiom (adj-x? (extent (pair-left p)) (extent (pair-right p)))

 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%% Template for displaying something on the screen

%% abstract-sort     a   a
%%                  / \ / \          equiv-rep?
%% rep-subsort      rrrrrrrr         rep?
%%                  ||||||||
%% concrete-sort  cccccccccccccc
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

spec GUI-TEMPLATE is
  sorts Abstract-Sort, Concrete-Sort
  sort-axiom Abstract-Sort = Concrete-Sort | rep? / equiv-rep?

  op rep : Abstract-Sort, Concrete-Sort -> Boolean
  op abs : Concrete-Sort | rep? -> Abstract-Sort

  op rep? : Concrete-Sort -> Boolean
  op equiv-rep? : Concrete-Sort | rep?, Concrete-Sort | rep? -> Boolean

  %% rep? picks outs the subsort of representatives

  definition of rep? is
    axiom (iff (rep? r) (ex (a) (rep a r)))
  end-definition

  %% any two representatives of an abstract thing are equivalent

  definition of equiv-rep? is
    axiom (iff
         (equiv-rep? r1 r2)
         (ex (a) (and (rep a ((relax rep?) r1))
                      (rep a ((relax rep?) r2)))))
  end-definition

  %% abs is the abstraction function for the quotient sort Abstract-Sort.
  %% Hence it is the "inverse" of rep.

  definition of abs is
    axiom (equal (abs r) ((quotient equiv-rep?) r))
  end-definition

  theorem abs-is-rep-inverse is
    (iff (rep a ((relax rep?) r)) (equal a (abs r)))

 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% GUI-TEMPLATE in which the concrete sort is a display object
```

```
spec TRIV-as-D-OBJECT is
  colimit of diagram
              nodes TRIV, DISPLAY-OBJECT, GUI-TEMPLATE
            arcs  TRIV -> DISPLAY-OBJECT : {E -> D-Object},
                  TRIV -> GUI-TEMPLATE   : {E -> Concrete-Sort}
          end-diagram

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ip-scheme Triv-to-D-Object : TRIV => DISPLAY-OBJECT is
  mediator TRIV-as-D-OBJECT
  dom-to-med {E -> Abstract-Sort}
  cod-to-med cocone-morphism from DISPLAY-OBJECT

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Given that there are schemes for representing the left and right
%% components of a pair, the representation of the pair itself imposes
%% the additional constraint that the chosen representatives for the
%% components be adjacent in the x-direction.

diagram PAIR-as-ADJ-X-PAIR--IMPORT-DIAGRAM is
  nodes body         : ADJ-X-PAIR,
        left-formal  : DISPLAY-OBJECT,
        right-formal : DISPLAY-OBJECT,
        shared-formal: REGION,
        left-actual  : TRIV-as-D-OBJECT,
        right-actual : TRIV-as-D-OBJECT,
        shared-actual: REGION,
      TRIV,
      gui-template-for-pair : GUI-TEMPLATE
  arcs  left-formal   -> body         : {D-Object -> Left-D-Object,
                                 extent -> left.extent},
        left-formal   -> left-actual  : cocone-morphism from DISPLAY-OBJECT,
        right-formal  -> body         : {D-Object -> Right-D-Object,
                                 extent -> right.extent},
        right-formal  -> right-actual : cocone-morphism from DISPLAY-OBJECT,
        shared-formal -> left-formal  : import-morphism,
        shared-formal -> right-formal : import-morphism,
        shared-actual -> left-actual  : {},
        shared-actual -> right-actual : {},
        shared-formal -> shared-actual: identity-morphism,
      %
      TRIV -> body                  : {E -> Adj-x-pair},
      TRIV -> gui-template-for-pair : {E -> Concrete-Sort}
 end-diagram

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

spec PAIR-as-ADJ-X-PAIR is
  import
  translate
  colimit of PAIR-as-ADJ-X-PAIR--IMPORT-DIAGRAM
  by {left-actual.Abstract-Sort  -> Left-Abstract-Sort,
      right-actual.Abstract-Sort -> Right-Abstract-Sort,
      gui-template-for-pair.Abstract-Sort -> Adj-x-Pair-Q}

  definition of gui-template-for-pair.rep is
    axiom (iff (rep abstract-pair region-pair)
               (and (rep (pi-1 abstract-pair) (pair-left region-pair))
```

```
                (rep (pi-2 abstract-pair) (pair-right region-pair))))
  end-definition

  op make-pair : Left-Abstract-Sort, Right-Abstract-Sort -> Adj-x-pair-Q
  op pi-1 : Adj-x-pair-Q -> Left-Abstract-Sort
  op pi-2 : Adj-x-pair-Q -> Right-Abstract-Sort

  %% non-constructive definition
  definition of make-pair is
    axiom (iff
         (equal (make-pair l r) p)
         (and (equal (pi-1 p) l)
              (equal (pi-2 p) r)))
  end-definition

  %%   L               P               R      abstract-sort
  %%  /|\             /|\             /|\       equiv-rep?
  %% / | \    pi-1   / | \   pi-2    / | \
  %% --L--  <------  --P--  ------>  --R--    rep-subsort
  %%   |       left     |     right     |       rep?
  %% --L--            --P--           --R--    concrete-sort

  definition of pi-1 is
    axiom (iff
         (equal (pi-1 ((quotient equiv-rep?) p)) ((quotient equiv-rep?) l))
         (equal (pair-left ((relax rep?) p)) ((relax rep?) l)))
  end-definition

  definition of pi-2 is
    axiom (iff
         (equal (pi-2 ((quotient equiv-rep?) p)) ((quotient equiv-rep?) r))
         (equal (pair-right ((relax rep?) p)) ((relax rep?) r)))
  end-definition

 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

interpretation
  PAIR-to-ADJ-X-PAIR : PAIR => ADJ-X-PAIR is
  mediator PAIR-as-ADJ-X-PAIR
  dom-to-med {L         -> Left-Abstract-Sort,
              R         -> Right-Abstract-Sort,
              Pair      -> Adj-x-pair-Q,
              make-pair -> make-pair,
              pi-1      -> pi-1,
              pi-2      -> pi-2}
  cod-to-med {pair.extent  -> body.pair.extent,
              left.extent  -> body.left.extent,
              right.extent -> body.right.extent}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% spec with example pair

spec PERSON is
  sort Person
  const P1 : Person
 end-spec
```

```
spec NAME is
  sort Name
  const N1 : Name
 end-spec

%% Observe the use of the empty specification to take the coproduct of
%% the actuals, PERSON and NAME. This is necessary because both these
%% specs will be refined into the same spec SAMPLE-LABELS.
%% The refinement of a colimit diagram produces a target spec defined
%% as the colimit of a diagram of the same shape. Had we not used the
%% empty spec, the refinement would have created two copies of
%% SAMPLE-LABELS.

spec EMPTY is end-spec

spec SAMPLE-PAIR is
  import
  translate
  colimit of diagram
              nodes TRIV1: TRIV, TRIV2: TRIV, PAIR, EMPTY,
                    PERSON, NAME
             arcs   TRIV1 -> PAIR   : {E -> L},
                    TRIV1 -> PERSON : {E -> Person},
                    TRIV2 -> PAIR   : {E -> R},
                    TRIV2 -> NAME   : {E -> Name},
                  EMPTY -> PERSON : {},
                  EMPTY -> NAME   : {}
           end-diagram
  by {L -> Person, R -> Name}

  const sample-pair : Pair

  definition of sample-pair is
    axiom (equal sample-pair (make-pair P1 N1))
  end-definition

 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Sample refinements for components of sample pair

spec SAMPLE-LABELS is
  import LABELLED-DISPLAY-OBJECT
  %% picture corresponding to sample person
  const P1-pic : Picture
  %% text corresponding to sample name
  const N1-txt : Text
 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

spec PERSON-as-SAMPLE-LABELS is
  import
  colimit of diagram
              nodes TRIV, SAMPLE-LABELS, GUI-TEMPLATE
             arcs   TRIV -> SAMPLE-LABELS : {E -> LD-Object},
                    TRIV -> GUI-TEMPLATE  : {E -> Concrete-Sort}
           end-diagram

  %% objects which represent persons are labelled with pictures
```

```
  axiom (implies (rep? x)
              (ex (pic) (equal (content (lbl x)) ((embed 1) pic))))

  %% two person-reps are equal iff they are labelled by the same picture

  definition of equiv-rep? is
    axiom (iff
         (equiv-rep? person-rep1 person-rep2)
         (equal (content (lbl ((relax rep?) person-rep1)))
                (content (lbl ((relax rep?) person-rep2)))))
  end-definition

  const P1-abs : Abstract-Sort

  %% any representation of sample person should be labelled by sample picture

  axiom (iff
       (rep P1-abs P1-rep)
       (ex (P1-mid)
        (and (equal P1-abs (abs P1-mid))
             (equal (content (lbl ((relax rep?) P1-mid)))
                    ((embed 1) P1-pic)))))

 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ip-scheme PERSON-to-SAMPLE-LABELS : PERSON => SAMPLE-LABELS is
  mediator PERSON-as-SAMPLE-LABELS
  dom-to-med {Person -> Abstract-Sort,
              P1     -> P1-abs}
  cod-to-med {LABEL.extent -> SAMPLE-LABELS.LABEL.extent,
            DISPLAY-OBJECT.extent -> SAMPLE-LABELS.DISPLAY-OBJECT.extent}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

spec NAME-as-SAMPLE-LABELS is
  import
  colimit of diagram
              nodes TRIV, SAMPLE-LABELS, GUI-TEMPLATE
            arcs  TRIV -> SAMPLE-LABELS : {E -> LD-Object},
                  TRIV -> GUI-TEMPLATE  : {E -> Concrete-Sort}
          end-diagram

  %% objects which represent names are labelled with text

  axiom (implies (rep? x)
              (ex (txt) (equal (content (lbl x)) ((embed 2) txt))))

  %% two name-reps are equal iff they are labelled by the same text

  definition of equiv-rep? is
    axiom (iff
         (equiv-rep? name-rep1 name-rep2)
         (equal (content (lbl ((relax rep?) name-rep1)))
                (content (lbl ((relax rep?) name-rep2)))))
  end-definition

  const N1-abs : Abstract-Sort
```

```
  %% any representation of sample name should be labelled by sample text

  axiom (iff
       (rep N1-abs N1-rep)
       (ex (N1-mid)
        (and (equal N1-abs (abs N1-mid))
             (equal (content (lbl ((relax rep?) N1-mid)))
                    ((embed 2) N1-txt)))))

 end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ip-scheme NAME-to-SAMPLE-LABELS : NAME => SAMPLE-LABELS is
  mediator NAME-as-SAMPLE-LABELS
  dom-to-med {Name -> Abstract-Sort,
              N1   -> N1-abs}
  cod-to-med {LABEL.extent -> SAMPLE-LABELS.LABEL.extent,
              DISPLAY-OBJECT.extent -> SAMPLE-LABELS.DISPLAY-OBJECT.extent}

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

ip-scheme
  EMPTY-to-LABELLED-DISPLAY-OBJECT : EMPTY => LABELLED-DISPLAY-OBJECT
  is
  mediator LABELLED-DISPLAY-OBJECT
  dom-to-med {}
  cod-to-med identity-morphism

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

A.5.2. Display Pair

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% A concrete representation of an abstract object.
%% ABS is surjective, REP is injective.

spec REP is
  sorts Abstract, Concrete

  op rep : Abstract -> Concrete
  op abs : Concrete -> Abstract

  axiom (equal (abs (rep a)) a)

  op abs? : Abstract, Concrete -> Boolean
  op rep? : Concrete, Abstract -> Boolean

  definition of abs? is
    axiom (iff (abs? a c) (equal a (abs c)))
  end-definition

  definition of rep? is
    axiom (equal (rep? c a) (abs? a c))
  end-definition

end-spec
```

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% Pair Homomorphisms

%%  SOURCE.Left <- SOURCE.Pair -> SOURCE.Right
%%       |              |              |
%%       V              V              V
%%  DEST.Left   <-  DEST.Pair  ->   DEST.Right

spec PAIR-HOMOMORPHISM is
  import SOURCE : PAIR, DEST : PAIR

  op left-map  : SOURCE.Left  -> DEST.Left
  op right-map : SOURCE.Right -> DEST.Right
  op pair-map  : SOURCE.Pair  -> DEST.Pair

  axiom (equal (left-map (SOURCE.left p)) (DEST.left (pair-map p)))
  axiom (equal (right-map (SOURCE.right p)) (DEST.right (pair-map p)))
end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% The above diagram, with both REP and ABS vertically.

spec REP-PAIR is
  translate colimit of diagram
  nodes
    PAIR-AC  : REP,
    LEFT-AC  : REP,
    RIGHT-AC : REP,

    REP-HOM : PAIR-HOMOMORPHISM,
    ABS-HOM : PAIR-HOMOMORPHISM,

    PAIR-REP  : ARROW, PAIR-ABS  : ARROW,
    LEFT-REP  : ARROW, LEFT-ABS  : ARROW,
    RIGHT-REP : ARROW, RIGHT-ABS : ARROW

  arcs
    PAIR-REP -> PAIR-AC : { f -> rep },
    PAIR-REP -> REP-HOM : { f -> pair-map },

    PAIR-ABS -> PAIR-AC : { f -> abs },
    PAIR-ABS -> ABS-HOM : { f -> pair-map },

    LEFT-REP -> LEFT-AC : { f -> rep },
    LEFT-REP -> REP-HOM : { f -> left-map },

    LEFT-ABS -> LEFT-AC : { f -> abs },
    LEFT-ABS -> ABS-HOM : { f -> left-map },

    RIGHT-REP -> RIGHT-AC : { f -> rep },
    RIGHT-REP -> REP-HOM  : { f -> right-map },

    RIGHT-ABS -> RIGHT-AC : { f -> abs },
    RIGHT-ABS -> ABS-HOM  : { f -> right-map }
  end-diagram by
    { PAIR-AC.Abstract -> Pair-Abstract,
      PAIR-AC.Concrete -> Pair-Concrete,
      LEFT-AC.Abstract -> Left-Abstract,
```

```
      LEFT-AC.Concrete -> Left-Concrete,
      RIGHT-AC.Abstract -> Right-Abstract,
      RIGHT-AC.Concrete -> Right-Concrete }

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

%% A ladder, ending in displayable pairs of adjacent display objects.

%%       OBJECT      ->      DISPLAY-OBJECT
%%         |                        |
%%         V                        V
%%        PAIR       ->      DISPLAY-PAIR
%%         |                        |
%%         V                        V
%%    PAIR-OF-DOS     ->  DISPLAY-PAIR-OF-DOS
%%         |                        |
%%         V                        V
%%  PAIR-OF-ADJ-DOS -> DISPLAY-PAIR-OF-ADJ-DOS

spec PAIR-OF-DOS is
  translate colimit of diagram
    nodes
      PAIR, REGION,
      LEFT : DISPLAY-OBJECT, RIGHT : DISPLAY-OBJECT
    arcs
      REGION -> LEFT  : import-morphism
      REGION -> RIGHT : import-morphism
      LEFT   -> PAIR  : { Display-Object -> Left }
      RIGHT  -> PAIR  : { Display-Object -> Right }
  end-diagram
  by { LEFT.Display-Object  -> Left-Display-Object,
       RIGHT.Display-Object -> Right-Display-Object }

spec PAIR-OF-ADJ-DOS is
  import translate PAIR-OF-DOS
  by { Pair-of-DOs -> Pair-of-Adj-DOs }
  axiom (adj-x? (extent (left p)) (extent (right p)))
end-spec

spec DISPLAY-PAIR is
  import translate colimit of diagram
    nodes OBJECT, PAIR, DISPLAY-OBJECT
    arcs
      OBJECT -> PAIR           : { Object -> Pair }
      OBJECT -> DISPLAY-OBJECT : { Object -> Display-Object }
  end-diagram
  by { Object -> Display-Pair }
end-spec

spec DISPLAY-PAIR-OF-DOS is
  import translate colimit of diagram
    nodes PAIR, DISPLAY-PAIR, PAIR-OF-DOs
    arcs
      PAIR -> DISPLAY-PAIR : { Pair -> Display-Pair }
      PAIR -> PAIR-OF-DOs  : { Pair -> Pair-of-DOs }
  end-diagram
  by { Pair -> Display-Pair-of-DOs }
  axiom (contains (extent p) (extent (left p)))
  axiom (contains (extent p) (extent (right p)))
end-spec
```

```
spec DISPLAY-PAIR-OF-ADJ-DOS is
  translate colimit of diagram
  nodes
    PAIR-OF-DOs, DISPLAY-PAIR, PAIR-OF-ADJ-DOs
  arcs
    PAIR-OF-DOs -> DISPLAY-PAIR     : { Pair-of-DOs -> Display-Pair }
    PAIR-OF-DOs -> PAIR-OF-ADJ-DOS : { Pair-of-DOs -> Pair-of-Adj-DOs }
  end-diagram
  by { Pair-of-DOs -> Display-Pair-of-Adj-DOs }
end-spec

%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Kestrel Institute

SPECWARE:™
Formal Support for Composing Software

by

Yellamraju V. Srinivas and Richard Jüllig

December 1994
Updated May 1995

Published in the Proceedings of the
Conference on Mathematics of Program Construction (Kloster Irsee, Germany, July 1995),
Lecture Notes in Computer Science, Vol. 947, Springer-Verlag, pp. 399–422.

KESTREL INSTITUTE • 3260 HILLVIEW AVENUE • PALO ALTO, CA 94304 • (415) 493-6871

SPECWARE:™
Formal Support for Composing Software

Yellamraju V. Srinivas and Richard Jüllig
Kestrel Institute
3260 Hillview Avenue
Palo Alto, CA 94304, USA
Email: {srinivas,jullig}@kestrel.edu
Tel: (415) 493-6871, Fax: (415) 424-1807

May 15, 1995

Abstract. SPECWARE supports the systematic construction of formal specifications and their stepwise refinement into programs. The fundamental operations in SPECWARE are that of composing specifications (via colimits), the corresponding refinement by composing refinements (via sheaves), and the generation of programs by composing code modules (via colimits). The concept of diagram refinement is introduced as a practical realization of composing refinements via sheaves. Sequential and parallel composition of refinements satisfy a distributive law which is a generalization of similar compatibility laws in the literature. SPECWARE is based on a rich categorical framework with a small set of orthogonal concepts. We believe that this formal basis will enable the scaling to system-level software construction.

Table of Contents

1 Introduction

SPECWARE™ supports the systematic construction of executable programs from axiomatic specifications via stepwise refinement. The immediate motivation for the the development of SPECWARE is the desire to integrate on a common conceptual basis the capabilities of several earlier systems developed at Kestrel Institute [Jüllig 93], including KIDS [Smith 90] and DTRE [Blaine and Goldberg 91].

1.1 Reasoning about the Structure of Specifications, Refinements, and Code

The most important new aspect of the framework developed is the ability to represent explicitly the structure of specifications, refinements, and program modules. We believe that the explicit representation and manipulation of structure is crucial to scaling program construction techniques to system development.

The basis of SPECWARE is a category of axiomatic specifications and specification morphisms. Specification structure is expressed via specification diagrams, directed multi-graphs whose nodes are labeled with specifications and arcs with specification morphisms. Specification diagrams are useful both for composing specification from pieces and for inducing on a given specification a structure suitable for the design task at hand.

In SPECWARE the design process proceeds by stepwise refinement of an initial specification into executable code. The unit of refinement is an interpretation, a theorem-preserving translation of the vocabulary of a source specification into the terms of a target specification. Each interpretation reduces the problem of finding a realization for the source specification to finding a realization for the target specification. The overall result of the design process is to refine an initial specification into a program module.

Of course, it is desirable to structure the overall refinement. Progression through multiple stages requires sequential composability of refinements. Similarly, parallel composition lets us exploit the structure of specifications by putting refinements together from refinements between sub-specifications of the source and target specifications. It is for this purpose that we introduce the notion of diagram refinements in this paper: just as specification diagrams impose a component structure on specifications, so do diagram refinements make explicit the component structure of a specification refinement.

Specification refinement exploits specification structure; code generation, in turn, exploits the refinement structure. Given translations to code for the specifications that serve as the final refinement targets, SPECWARE generates a system of modules by induction on the refinement structure. Layered module construction mirrors sequential composition of refinements, and the "gluing together" of modules into larger modules reflects the (parallel) composition of specifications and refinements from components.

Our work combines ideas and notions from the fields of algebraic specifications, category theory, and sheaf theory. We believe that the use of such "heavy" formal machinery is well-justified. For instance, category theory seems ideally suited for describing the manipulation of richly detailed structures at various levels of granularity. Similarly, the sheaf-theoretic notion

of compatible families seems fundamental to and pervasive in putting systems together from interdependent components.

The ideas and concepts presented in this paper have been implemented in the SPECWARE 1.0 system, which continues to be developed. It is interesting to note that the implementation efforts seem to fare the better the more closely the implementation reflects the underlying theoretical concepts. Conversely, experimentation with the SPECWARE system has had a significant impact on the theory of diagram refinement presented here.

1.2 Outline

We briefly present our specification language in Sect. 2 and in Appendix A. The focus of this paper is the sequential and parallel composition of refinements, as described in Sect. 4. Sect. 5 discusses how sufficiently refined specifications can be translated to programs. Sect. 6 describes related work. Finally, we offer some conclusions and an outlook on future work.

2 Putting Specifications Together

The primary component of the SPECWARE workspace is the category of specifications and specification morphisms. Diagrams in this category describe system structure. Specifications can be put together via colimits to obtain more complex specifications. We will only briefly describe these concepts because these ideas are well known; see, e.g., [Burstall and Goguen 77, Sannella and Tarlecki 88a].

2.1 Specifications

A *specification* is a finite presentation of a theory in higher-order logic. An uncommon feature of SPECWARE is that subsorts and quotient sorts can be defined using predicates and equivalence relations, respectively. For details of the particular logic used, see Appendix A.

2.1.1 Specification-Constructing Operations

Specifications can either be directly given (as a set of sorts, operations, axioms, etc.) or constructed from other specifications via the following operations (inspired by ASL [Wirsing 86, Sannella and Tarlecki 88a])

translate ⟨spec⟩ **by** ⟨renaming-rules⟩
colimit of ⟨diagram⟩
spec import ⟨spec⟩ ⟨spec-elements⟩ **end-spec**

"Translate" creates a copy of a specification with some elements renamed according to the given renamings; an isomorphism is also created between the original and the translated specifications. "Colimit" is the standard operation from category theory (see, e.g., [Mac Lane 71]); colimits are constructed using equivalence classes of sorts, operations, etc.

"Import" places a copy of the imported specification[1] in the importing specification; an inclusion morphism is also generated.

2.2 Specification Morphisms

A *specification morphism* (or simply a *morphism*) translates the language of one specification into the language of another specification in a way that preserves theorems. Specification morphisms underlie almost all constructions in SPECWARE.

2.2.1 Flavors of Specification Morphisms

The set of sorts given in a specification generates a free algebra via sort-constructing operations such as product, coproduct, etc. A specification morphism is a map from the sorts[2] and operations of one specification to the sorts and operations of another such that (1) the map is a homomorphism on the sort algebras, (2) the ranks of operations are translated compatibly with the operations, and (3) axioms are translated to theorems.

A presentation of a specification morphism in SPECWARE is a finite map from the declared sorts in the source specification to the declared or constructed sorts in the target specification, and from source operations to target operations, such that the map generates a specification morphism as described above.

Many flavors of morphisms can be defined for specifications, ranging from axiom-preserving presentation morphisms to logical morphisms between the toposes (theories) generated by the source and target specifications. The choice made in SPECWARE (declared sorts mapping to constructed sorts) is a pragmatic one, a compromise between simplicity and flexibility—morphisms are simple enough for use in putting specifications together, while flexible enough to model refinement.

2.3 Specification Diagrams

A morphism from A to B may be construed as indicating how A is a "part of" B. Thus, we can use morphisms to express a system as an interconnection of its parts, i.e., as a diagram. Formally, a *diagram* is a directed multigraph in which the nodes are labeled by specifications, and the edges by specification morphisms (in a multigraph, there can be more than one edge between any two nodes).[3]

2.3.1 Composition (Putting Specifications Together)

We can reduce a diagram of specifications to a single specification by taking the colimit of the diagram. The colimit of a diagram is constructed by first taking the disjoint union (coproduct) of all the specifications in the diagram and then the quotient of this coproduct via the equivalence relation generated by the morphisms in the diagram. The result will be

[1] Only one specification can be imported. A colimit is necessary if multiple specifications are to be imported.

[2] Here, we take "sorts" to mean all the sorts in the sort algebra.

[3] When convenient, we will treat a diagram as a functor from the category freely generated by its underlying graph to the category of specifications and specification morphisms.

a valid specification (i.e., the colimit exists) only if the sort algebra is free (this means that two structurally dissimilar sorts cannot be identified in a colimit).

Example 1. The specifications for topological sorting are shown in Fig. 1 (following Knuth [Knuth 68, pp. 258–265]). The problem of topological sorting is specified as an input-output relation. To specify this relation, we need the concepts of partial order and total order on some set of elements; these specifications are first put together via a colimit and then imported. The specification for partial orders contains a membership predicate and a less-or-equal predicate with appropriate axioms. The specification for total orders renames the partial orders specification and extends it with a totality axiom and a less-than predicate.

In the figure, the arrow labeled "d" is a definitional extension and the arrows labeled "c" are part of a colimit cocone.

3 Stepwise Refinement

The development process of SPECWARE is intended to support the refinement of a problem specification into a solution specification. Refinements introduce additional design detail, e.g., the transformation of definitions into constructive definitions, representation choices for data types, etc. SPECWARE's refinement constructs, introduced below, address three important aspects of refinement:

problem reduction: construction of a solution relative to some base;

stepwise refinement: sequential composition of refinements; and

putting refinements together: parallel composition of refinements.

3.1 Interpretations

The notion of refinement in SPECWARE is that a specification B refines a specification A if there is a construction which produces models of A from models of B [Sannella and Tarlecki 88b]. Specification morphisms serve this purpose because associated with every morphism $\sigma : A \rightarrow B$ there is a reduct functor $\_|_\sigma$ which produces models of A from models of B. Morphisms, however, are too weak to represent refinements which normally occur during software development. So, we use a more general notion, *interpretations*, which are specification morphisms from the source specification to a definitional extension of the target specification.

Definition 1 (*Interpretation*). An *interpretation* $\rho : A \Rightarrow B$ from a specification A (called *domain* or *source*) to a specification B (called *codomain* or *target*) is a pair of morphisms $A \rightarrow A\text{-}as\text{-}B \leftarrow B$ with common codomain $A\text{-}as\text{-}B$ (called *mediating* specification or simply *mediator*), such that the morphism from B to $A\text{-}as\text{-}B$ is a definitional extension.

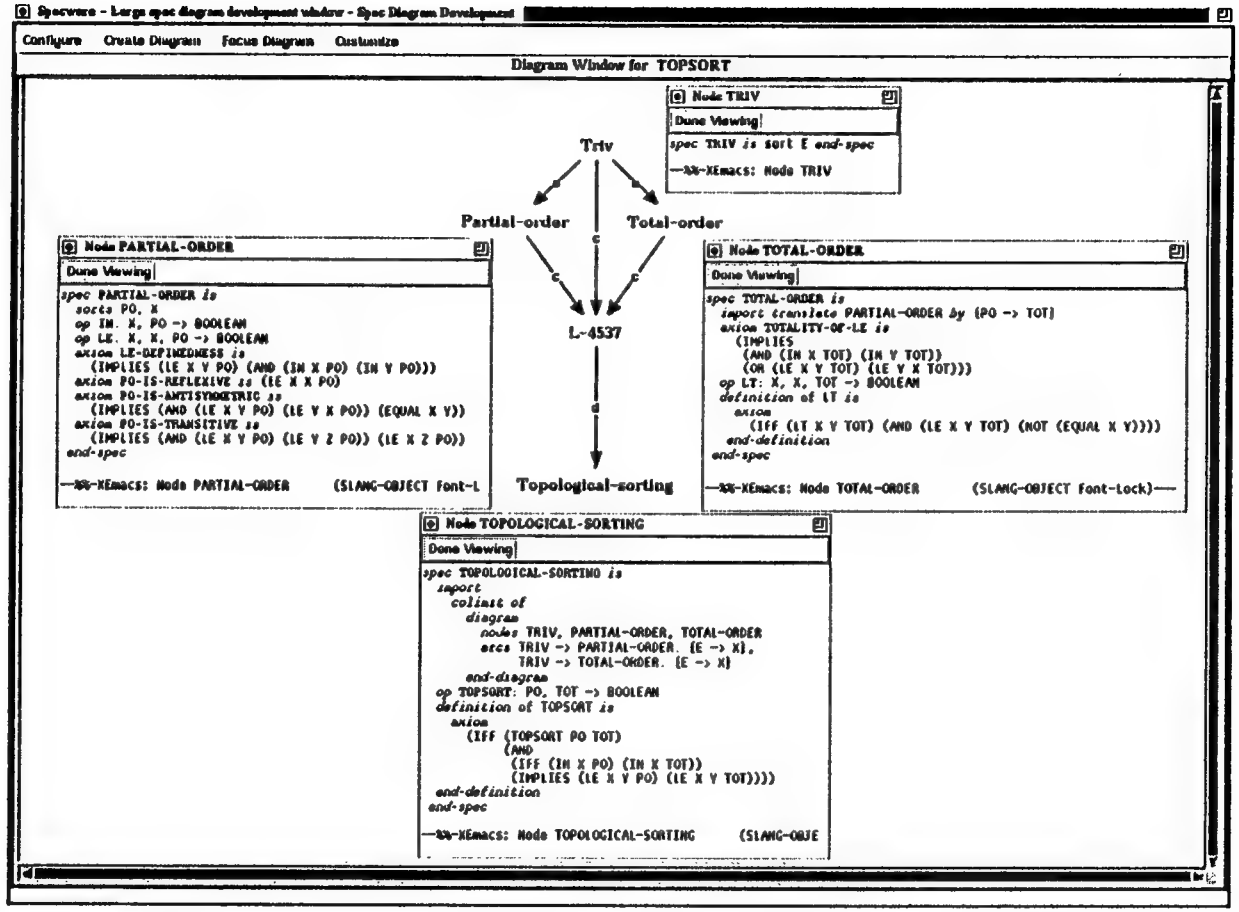


Fig. 1. Specification for topological sorting

Definition 2 (*Definitional extension*). A morphism $S \to T$ is a *strict definitional extension* if it is injective and if every element of T which is outside the image of the morphism is either a defined sort or a defined operation. A *definitional extension* is a strict definitional extension optionally composed with a specification isomorphism.

In this case, we also sometimes say that T is a definitional extension of S. Definitional extensions are indicated in diagrams by $\dashrightarrow$.

A specification and any definitional extension of it generate the same topos (or theory). Hence, interpretations are generalized morphisms. Interpretations are a suitable notion of refinement because models of the source specification can be constructed from models of the target specification by first expanding them along the definitional extension and then taking reducts.

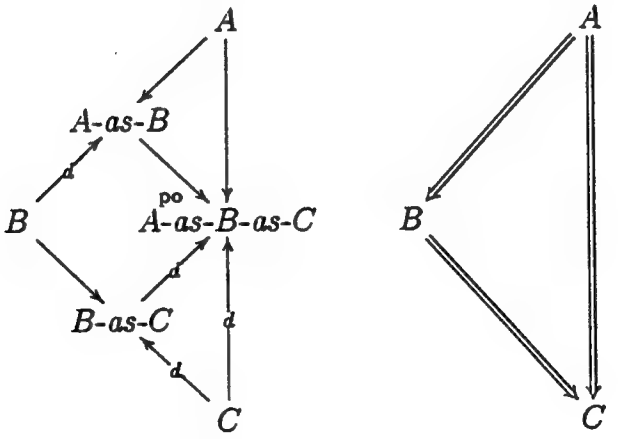
Example 2. We show in Fig. 2 an interpretation from total orders to sequences in which total orders are represented as a subsort of sequences: a sequence represents a total order if and only if it does not contain any duplicate elements. This subsort is defined in the mediating specification. Total-order operations are then defined on this subsort in terms of the underlying sequence operations.

In general, a source sort may be represented by a more elaborately constructed sort. For example, partial orders can be represented as a quotient of a subsort of graphs: to qualify as a representative, a graph must be acyclic (this is the subsort predicate), and two acyclic graphs represent the same partial order if their transitive closure is the same (this is the equivalence relation for the quotient sort).

Interpretations encompass and generalize the data type refinement introduced in [Hoare 72] and other similar schemes.

3.2 Sequential (Vertical) Composition of Interpretations

Given two interpretations $\rho_1 : A \Rightarrow B$ and $\rho_2 : B \Rightarrow C$ such that the codomain of the first is the domain of the second, their sequential composition $\rho_2 \circ \rho_1 : A \Rightarrow C$ is obtained as in the diagram below (the marking "po" indicates a pushout square).[4] We use the facts that definitional extensions are closed under composition and are preserved by pushouts.



[4] Diagrams are assumed to be commutative unless stated otherwise.

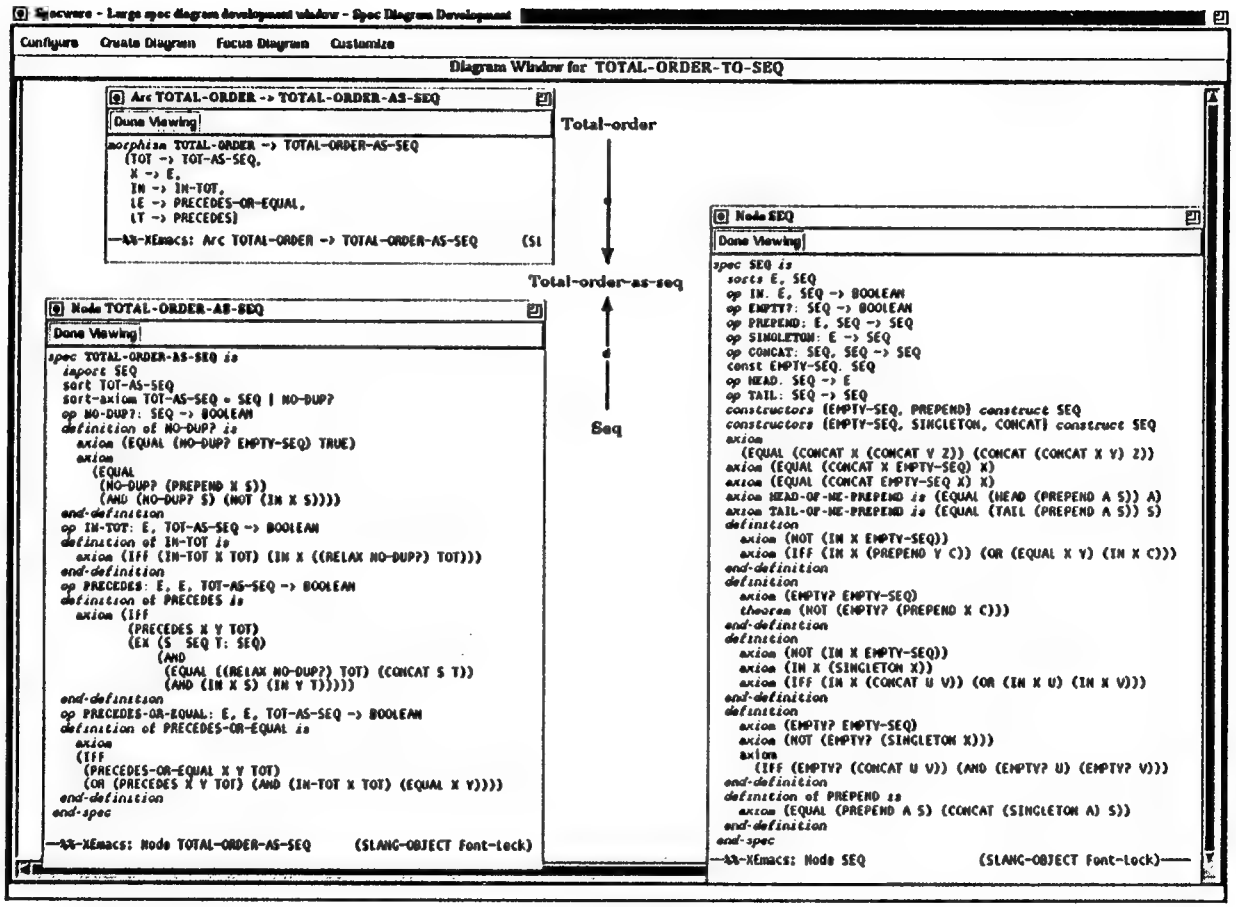


Fig. 2. An interpretation of total orders as a subsort of sequences

Sequential composition of interpretations facilitates incremental, layered refinement.

3.3 Algorithm Synthesis and Interpretation Construction

Algorithm synthesis plays two roles in the model of software development supported by SPECWARE:

- the creation of constructive definitions in interpretations, and
- the refinement of input-output relations sufficient to extract a constructively defined function.

Note that the definitions used in the mediating specification of an interpretation are not required to be constructive. As an example, see the definition of PRECEDES in the specification TOTAL-ORDER-AS-SEQ in Fig. 2. If we want to generate code corresponding to this operation, then we have to further refine this definition, with the goal of replacing the existential quantifier by an algorithm.

Similarly, the input-output relations used in a top-level specification are not usually functional. As an example, see the definition of the relation TOPSORT in the specification TOPOLOGICAL-SORTING in Fig. 1. If we want to find a function which satisfies this relation, we have to further refine the enclosing specification. This refinement can be guided by a hierarchy of algorithm theories which are used to impose additional structure on the specification. Details of this process can be found in [Smith 93, Smith and Lowry 90].

Algorithm synthesis is one of the creative parts of software development and can be used to construct basic interpretations which can then be composed. SPECWARE aids this by providing a scaffolding which takes care of the mundane details, thus letting the developer identify and focus on the creative part.

4 Putting Refinements Together

Just as a specification can be put together from smaller specifications, so can refinements of a specification be put together from refinements of component specifications. Formally, the various ways of constructing specifications generate a Grothendieck topology on the category of specifications and specification morphisms, and refinements form a sheaf with respect to this topology. Introductions to Grothendieck topologies and sheaves can be found in [Mac Lane and Moerdijk 92], [Artin et al. 72, Exposés I–IV]; an application to algorithm derivation and several computer science examples can be found in [Srinivas 93].

4.1 Theoretical Basis: A Sheaf of Refinements

Definition 3 (*A Topology for Specifications*). We obtain a Grothendieck topology on the category of specifications and specification morphisms by defining a family of specification morphisms $\{ S_i \rightarrow S \}$ with common codomain to be a covering family if S is a definitional extension of the union of the images of the arrows in the family.

Definition 4 (*Image of a Specification Morphism*). The image of a specification morphism $\sigma : S \rightarrow T$ is the specification consisting of all elements $\sigma(x)$ where x is any element of the source specification, e.g., sort, operation, theorem, etc.

To see that the topology above encompasses the specification constructing operations of Section 2.1.1, observe that a translation generates an isomorphism (which is a singleton covering family), and that a colimit specification is covered by its family of cocone arrows. The case of import can be reduced to that of colimit. However, it is useful to distinguish the case when the import morphism is a definitional extension; it then forms a (singleton) covering family.

Given any cover for a specification, a refinement for the specification can be constructed from refinements for the elements of the cover, provided the refinements are "compatible". This observation leads to a sheaf.

Definition 5 (*A Sheaf of Refinements*). Assume a fixed specification B, the base specification. Define a functor $\mathcal{R} : \mathbf{Spec}^{\mathbf{op}} \rightarrow \mathbf{Set}$ by assiging to each specification S the set of all interpretations (refinements) from S to B, and to each specification morphism $m : S \rightarrow T$ the function which restricts an interpretation $\rho : T \Rightarrow B$ to an interpretation $\rho \circ m : S \Rightarrow B$. This functor is a sheaf with respect to the Grothendieck topology defined above.

The sheaf condition asserts that for every cover $\{ f_i : S_i \rightarrow S \mid i \in I \}$, every compatible family of interpretations $\{ \rho_i : S_i \Rightarrow B \mid i \in I \}$ can be uniquely extended to an interpretation $\rho : S \rightarrow B$ such that the restriction of ρ along any f_i is equal to ρ_i.

Informally, a family of interpretations $\{ \rho_i : S_i \Rightarrow B \mid i \in I \}$ is compatible if the member interpretations agree wherever the pieces of the cover overlap. In this case, an interpretation $\rho : S \rightarrow B$ can be constructed as the shared union of the given family of interpretations. The details of this construction will be omitted here, because the construction is similar to the parallel composition of interpretations described below.

4.2 Practical Realization: Diagram Refinement

Three factors prevent a direct realization of the sheaf-theoretic view of putting interpretations together presented in the previous section: (1) The compatibility condition is hard to check because pullbacks do not exist in general in the category of specification morphisms; (2) Equality of interpretations is hard to check; (3) It is unrealistic to assume that a single base specification (the refinement target) is given. Typically, we would like to assemble a target specification as we refine pieces of the source specification.

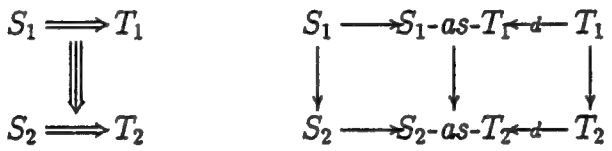
We handle (1) by using only those covers which are directly given by specification construction operations. In particular, a (finite) colimit explicitly indicates the shared parts among the components of a specification. (2) is handled by introducing interpretation morphisms, which explicitly indicate how one interpretation specializes another. We also use a strong equality for morphisms which can be checked syntactically; see Definition 6 below. (3) is handled by using diagrams in the category of interpretations and interpretation morphisms. A preliminary target specification can be assembled from the codomains of the

interpretations in a diagram. The target specification can be further modified by modifying the diagram of specifications that defines it.

We will describe these concepts below, finally obtaining a notion of refinement for diagrams.

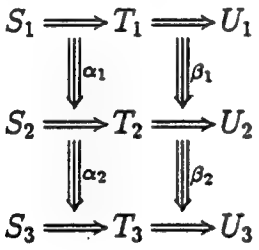
Definition 6 (*Strong Morphism Equality*). Two specification morphisms $\sigma, \tau : S \to T$ are equal if for each sort or operation $x \in S$, $\sigma(x) = \tau(x)$.

Definition 7 (*Interpretation Morphism*). An interpretation morphism from an interpretation $\rho_1 : S_1 \Rightarrow T_1$ to another interpretation $\rho_2 : S_2 \Rightarrow T_2$ is a triple of specification morphisms such that the diagram on the right below commutes.



Interpretations and interpretation morphisms form a category **Interp**. Another view of this category is as (a sub-category of) the functor category of functors from $\bullet \to \bullet \leftarrow \bullet$ to the category **Spec** of specifications and specification morphisms. Hence, colimits in **Spec** lift to colimits in the category of interpretations.[5]

Specifications, interpretations, and interpretation morphisms form a double category. That is, in addition to the obvious sequential/vertical composition of interpretation morphisms, there is also a parallel/horizontal composition of interpretation morphisms. The two compositions satisfy an interchange law: given six interpretations and four interpretation morphisms as shown on the left below, the equation on the right is true.



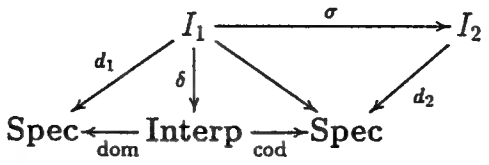
$$(\beta_2 \bullet \alpha_2) \circ (\beta_1 \bullet \alpha_1) = (\beta_2 \circ \beta_1) \bullet (\alpha_2 \circ \alpha_1)$$

Now, given two specifications which are defined as colimits, a compatible family of interpretations can be given as a diagram of interpretations. It will be useful here to treat diagrams as functors.

Definition 8 (*Diagram Refinement*). Given two diagrams of specifications $d_1 : I_1 \to \mathbf{Spec}$ and $d_2 : I_2 \to \mathbf{Spec}$, a diagram refinement $\langle \delta, \sigma \rangle : d_1 \to d_2$ is a pair consisting of a diagram of interpretations $\delta : I_1 \to \mathbf{Interp}$ with shape I_1 and a functor $\sigma : I_1 \to I_2$ between the two shapes such that the following diagram commutes (dom and cod are the obvious functors

[5] Definitional extensions are preserved by colimits.

which maps interpretations and interpretation morphisms to their domains and codomains, respectively).



Example 3. In Fig. 3, we show a refinement of the specification for topological sorting (shown in Fig. 1): the partial orders are refined to pairs of sequences (one listing the elements and another listing the ordering relation), and the total orders are refined to sequences (as shown in Fig. 2).

The components of the colimit which defines the import into the specification for topological sorting are refined in parallel. The vertical interpretations emanating from this diagram form a diagram refinement. Note that the target diagram has a shape which is different from that of the source diagram: the extra arrow in the target diagram is used to identify the sequences which represent the elements of the partial orders and the total orders (remember that topological sorting takes as input a partial order and produces a total order on the *same* set of elements).

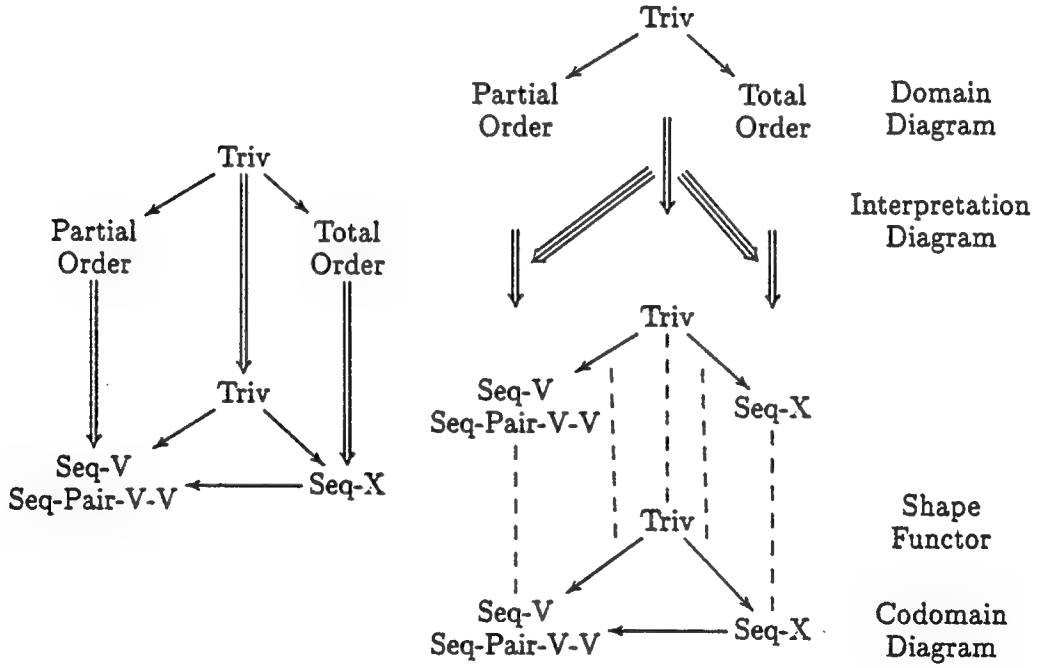
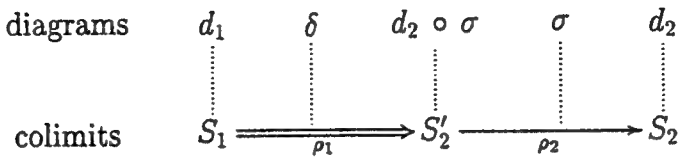


Fig. 3. Components of a diagram refinement

4.2.1 Parallel (Horizontal) Composition of Interpretations

As expected, a diagram refinement yields a refinement from the colimit of the source diagram to the colimit of the target diagram. Consider the diagram refinement $\langle \delta, \sigma \rangle : d_1 \rightarrow d_2$ above.

Let S_1 and S_2 be the colimits of the two diagrams. The colimit of the interpretation diagram δ is an interpretation $\rho_1 : S_1 \Rightarrow S_2'$ from S_1 to the colimit (say S_2') of the diagram $d_2 \circ \sigma : I_1 \rightarrow \mathbf{Spec}$. The colimit cocone $d_2 \stackrel{\cdot}{\rightarrow} S_2$ when composed with the shape morphism σ gives a cocone $d_2 \circ \sigma \stackrel{\cdot}{\rightarrow} S_2$. From this, we obtain a witness arrow $\rho_2 : S_2' \rightarrow S_2$. The composition $\rho_2 \circ \rho_1$ is the desired parallel composition of the diagram refinement $\langle \delta, \sigma \rangle : d_1 \rightarrow d_2$.

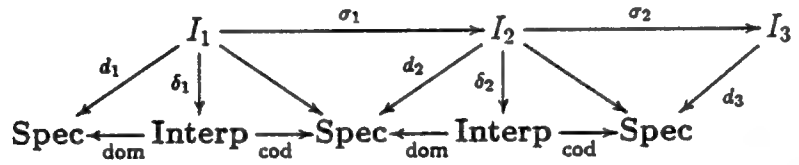


We will denote the parallel composition of a diagram refinement Δ by $|\Delta|$.

Example 4. In Fig. 4, we show details of the refinement of the specification for topological sorting. The figure illustrates both sequential and parallel composition of interpretations. As an example of sequential composition, partial orders are refined to pairs of sequences by representing them as graphs; the graphs are then represented as sets of nodes and sets of edges; then, these sets are represented as sequences. There are also several parallel compositions, e.g., the refinements of Set-of-Pair and TS-Import.

4.2.2 Composing Diagram Refinements

Diagram refinements can be composed by composing the individual interpretations which comprise them. Let $\langle \delta_1, \sigma_1 \rangle : d_1 \rightarrow d_2$ and $\langle \delta_2, \sigma_2 \rangle : d_2 \rightarrow d_3$ be two diagram refinements. We can juxtapose these as shown below.



Now, as shown below, we get two diagrams of interpretations with shape I_1, namely δ_1 and $\delta_2 \circ \sigma_1$, such that the codomains of the interpretations in the first diagram match with the domains of the interpretations in the second diagram. By composing the individual interpretations, we get another interpretation diagram with shape I_1. We will denote this horizontally composed diagram of interpretations by $(\delta_2 \circ \sigma_1) \bullet \delta_1$. The shape morphism for the composed diagram refinement is obtained by composing the individual shape morphisms, $\sigma_2 \circ \sigma_1 : I_1 \rightarrow I_2 \rightarrow I_3$. Thus, $\langle (\delta_2 \circ \sigma_1) \bullet \delta_1, \sigma_2 \circ \sigma_1 \rangle : d_1 \rightarrow d_3$ is the composition of the two diagram refinements we started with.

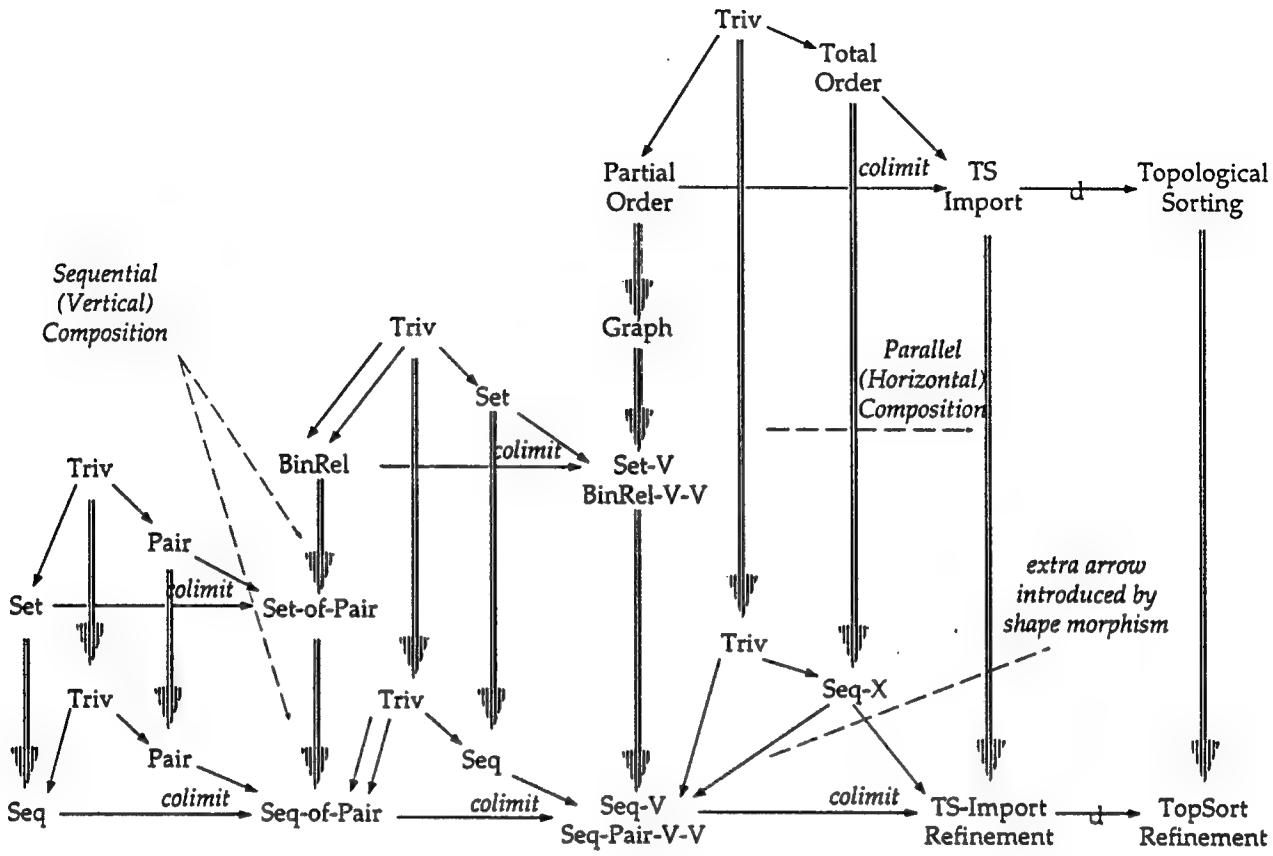
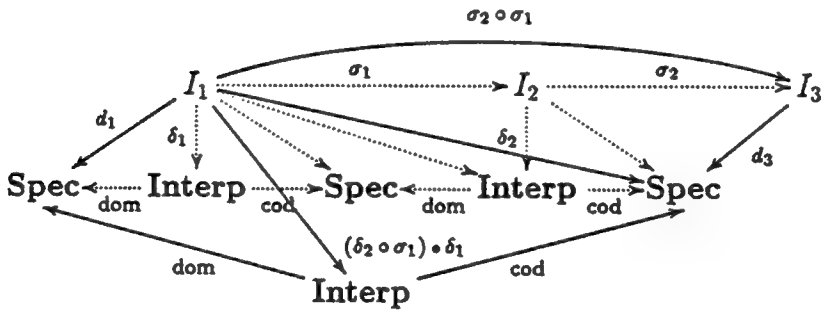


Fig. 4. Refinement of topological sorting



4.2.3 Compatibility of Vertical and Horizontal Interpretation Composition

If $\Delta_1 : d_1 \to d_2$ and $\Delta_2 : d_2 \to d_3$ are two diagram refinements which can be composed, then the distributive law satisfied by them is

$$|\Delta_2| \circ |\Delta_1| = |\Delta_2 \circ \Delta_1|.$$

This can be verified by straighforward diagram chasing (using the interchange law for interpretation morphisms). Thus, $|\_|$ is a functor from the category of diagrams and diagram refinements to the category of specifications and interpretations.

The distributive law above is a generalization of other such laws introduced in the literature. The law introduced by Goguen and Burstall [Goguen and Burstall 80] is too constraining to be practically useful. The law introduced by Sannella and Tarlecki [Sannella and Tarlecki 88b] uses parameterization and does not handle colimits; moreover, it is semantically oriented.

5 Putting Code Fragments Together

When specifications are sufficiently refined, they can be converted into programs which realize them. This involves a switching of logics. We use the theory of logic morphisms described by Meseguer [Meseguer 89]. We will confine our attention to entailment systems and their morphisms, rather than logics (which include models and institutions). Entailment systems are sufficient for the purpose of code generation.

5.1 Entailment Systems and their Morphisms

Definition 9 (*Entailment System*). An entailment system is a triple $\langle \mathbf{Sig}, sen, \vdash \rangle$ consisting of

1. a category **Sig** of signatures and signature morphisms,
2. a functor $sen : \mathbf{Sig} \to \mathbf{Set}$ (where **Set** is the category of sets and functions) which assigns to each signature Σ the set of Σ-sentences, and to each signature morphism $\sigma : \Sigma \to \Sigma'$, the function which translates Σ-sentences to Σ'-sentences (this function will also be denoted by σ), and
3. a function $\vdash$ which associates to each signature Σ a binary relation $\vdash_\Sigma \subseteq \mathcal{P}(sen(\Sigma)) \times sen(\Sigma)$, called Σ-entailment,

such that the following properties are satisfied:

1. *reflexivity:* for any $\varphi \in sen(\Sigma)$, $\{\varphi\} \vdash_\Sigma \varphi$;
2. *monotonicity:* if $\Gamma \vdash_\Sigma \varphi$ and $\Gamma' \supseteq \Gamma$, then $\Gamma' \vdash_\Sigma \varphi$
3. *transitivity:* if $\Gamma \vdash_\Sigma \varphi_i$, for $i \in I$, and $\Gamma \cup \{\varphi_i \mid i \in I\} \vdash_\Sigma \psi$, then $\Gamma \vdash_\Sigma \psi$;
4. $\vdash$*-translation:* if $\Gamma \vdash_\Sigma \varphi$, then for any signature morphism $\sigma : \Sigma \to \Sigma'$, $\sigma(\Gamma) \vdash_{\Sigma'} \sigma(\varphi)$.

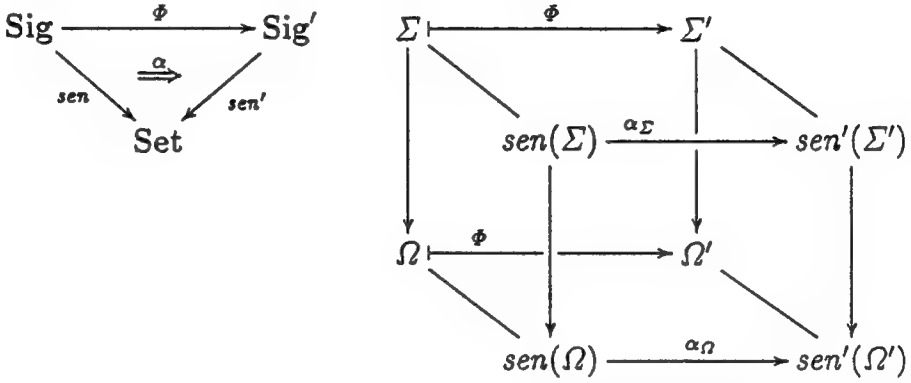
To map one entailment system into another, we map the syntax (i.e., signatures and sentences) while preserving entailment. Preservation of entailment represents the relevant correctness criterion for translating specifications from one logic to another. Note that this is similar to the correctness criterion for refinement within a single logic.

A simple way to map syntax is to map signatures to signatures, and sentences over a signature to sentences over the translated signature. If the former is a functor, the latter becomes a natural transformation.

Definition 10 (*Entailment system morphism—plain version*). A morphism between entailment systems $\langle \Phi, \alpha \rangle : \langle \mathrm{Sig}, sen, \vdash \rangle \to \langle \mathrm{Sig}', sen', \vdash' \rangle$ is a pair consisting of a functor $\Phi : \mathrm{Sig} \to \mathrm{Sig}'$ which maps signatures to signatures and a natural transformation $\alpha : sen \dot{\to} sen' \circ \Phi$ which maps sentences to sentences such that entailment is preserved:

$$\Gamma \vdash_\Sigma \varphi \Rightarrow \alpha_\Sigma(\Gamma) \vdash'_{\Phi(\Sigma)} \alpha_\Sigma(\varphi).$$

We can visualize α and the naturality condition in the following diagrams.



Morphisms which map signatures to signatures are not flexible enough, especially for code generation. In general, it may be necessary to map built-in elements of one logic into defined elements of another, and vice versa. This can be realized by mapping signatures to specifications, and vice versa, or, in general, specifications to specifications.

However, morphisms which map specifications to specifications are too unconstrained. So Meseguer [Meseguer 89] proposes a general version of entailment system morphisms which map specifications to specifications "sensibly". We will use these morphisms but omit the detailed definition here.

5.2 Translating from Slang to Lisp

The specification language used in SPECWARE is called SLANG. We distinguish SLANG because SPECWARE may have multiple back-ends, Lisp, C, Ada, etc., each with its own logic.

We consider a sub-logic of SLANG, called the *abstract target language* (for LISP); there is one sub-logic for each language into which SLANG specifications can be translated. We will denote this sub-logic by SLANG^{--}. The sub-logic SLANG^{--} is defined by starting with a set of basic specifications, such as integers, sequences, etc., which have direct realizations in the target language. All specifications which can be constructed from the base specifications, with the following restrictions, are then included in the sub-logic:

– for colimit specifications, only injective morphisms are allowed in the diagram;[6]

[6] For colimit specifications which can be construed as "instantiations" of a "generic" specification, the morphisms from the formal to the actual may be non-injective.

– all definitions must be constructive, i.e., they must either be explicit definitions (e.g., `(equal (square x) (times x x))`), or, if they are recursive, they must be given as conditional equations using a constructor set.

The goal of the refinement process is to arrive at a sufficiently detailed specification which satisfies the restrictions above.

The sub-logic SLANG^{--} will be translated into a functional subset of LISP. To facilitate this translation, we couch this subset as an entailment system, denoted LISP^{--}. The signatures of this entailment system are finite sets of untyped operations and the sentences are function definitions of the form

```
(defun f (x)
  (cond ((p x) (g x))
   ...))
```

and generated conditional equations of the form

```
(if (p x) (equal (f x) (g x))).
```

The entailment relation is that of rewriting, since theories in LISP^{--} can be viewed as conditional-equational theories over the simply-typed λ-calculus.

In Fig. 5, we show a fragment of an entailment system morphism from SLANG^{--} to LISP^{--}. Note, in particular, the translations from and to empty specifications. The set of sentences in the SLANG specification `INT` translates to the empty set; this is because integers are primitive in LISP. Similarly, the empty SLANG specification translates to a non-empty LISP specification; this is because some built-in operations of SLANG are not primitive in LISP.

5.2.1 Translating Constructed Sorts

There are numerous details in entailment system morphisms such as that from SLANG^{--} to LISP^{--}. We will briefly consider the translation of constructed sorts. Subsorts can be handled by representing elements of a subsort by the corresponding elements of the supersort. Similarly, quotient sorts can be handled by representing their elements by the elements of the base sort. Sentences have to be translated consistently with such representation choices: e.g., injections associated with subsorts (`(relax p)`) and the surjections associated with quotient sorts (`(quotient e)`) must be dropped. Also, the equality on a quotient sort must be replaced by the equivalence relation defining the quotient sort.

In Fig. 6, we show the representation of coproduct sorts by variant records. This translation exploits the generality of entailment system morphisms: a signature is mapped into a theory.

5.3 Translation of Colimits: Putting Code Fragments Together

If an entailment system morphism is defined in such a way that it is co-continuous, i.e., colimits are preserved, then we obtain a recursive procedure for translation, which is similar

SLANG^{--}	$\longrightarrow$ LISP^{--}
EMPTY	$\longmapsto$ spec SLANG-BASE is ops implies, iff (defun implies (x y) (or (not x) y)) (defun iff (x y) (or (and x y) (and (not x) (not y)))) end-spec
INT	$\longmapsto$ SLANG-BASE
spec FOO is import INT op abs : Int -> Int definition of abs is axiom (implies (ge x zero) (equal (abs x) x)) axiom (implies (lt x zero) (equal (abs x) (minus zero x))) end-definition end-spec	$\longmapsto$ spec FOO' is import SLANG-BASE op abs (defun abs (x) (cond ((>= x 0) x) ((< x 0) (- 0 x)))) end-spec

Fig. 5. Fragment of entailment system morphism from SLANG^{--} to LISP^{--}

```
spec STACK is                          ⟼  spec STACK' is
import INT                                 import SLANG-BASE
...                                        op size, E-Stack?, NE-Stack?
sort-axiom                                 (defun E-Stack? (s)
  Stack = E-Stack + NE-Stack                (= (car s) 1))
...                                        ...
op size  : Stack -> Int                    (defun size (s)
definition of size is                       (cond
 axiom                                       ((E-Stack? s) 0)
 (equal (size ((embed 1) s))                 ((NE-Stack? s)
        zero)                                 (1+ (size (pop (cdr s)))))
 axiom                                      ))
 (equal (size ((embed 2) s))               end-definition
        (succ (size (pop s))))             end-spec
end-definition
end-spec
```

Fig. 6. The representation of coproduct sorts as variant records

to that of refinement: the code for a specification can be obtained by assembling the code for smaller specifications which cover it.

The entailment system morphism from SLANG^{--} to LISP^{--} briefly described above does preserve colimits because of our restriction to injective morphisms. In general, this is true for most programming languages because they only allow imports, which are inclusion morphisms.

6 Related Work

SPECWARE builds upon a large body of work in formal specifications and program synthesis and transformation developed over the last two decades.

The design of SLANG, the specification language of SPECWARE, was inspired by Sanella and Tarlecki's [Sannella and Tarlecki 88a] and Wirsing's work [Wirsing 86] on structured algebraic specifications. Putting theories together via colimits was first proposed by Burstall and Goguen as part of CLEAR [Burstall and Goguen 77]. SLANG was further influenced by CIP [Bauer et al. 85, Bauer et al. 87] and OBJ [Goguen and Winkler 88].

SPECWARE adopts in a higher-order setting the notion of interpretations as refinements from Turski's and Maibaum's development in first-order logic [Turski and Maibaum 87]. SPECWARE could be construed as a realization of the design methodology espoused by Lehman, Stenning, and Turski, with the addition of parallel refinement composition [Lehman et al. 84]. The notion of parallel refinement composition described in this paper is different from the horizontal composition of parameterized specifications described by Sannella and Tarlecki [Sannella and Tarlecki 88b].

The explicit use of subsort and quotient sort constructions in SPECWARE connects data type refinement in an algebraic setting with Hoare's abstraction/refinement functions [Hoare 72] which also underlie the refinement found in VDM [Jones 86].

Our work is both similar and complementary to Bird's and Meertens' equational reasoning approach to program development [Bird 86, Bird 87]. Reasoning about commuting specification diagrams is equational reasoning at the specification level; Bird's and Meertens' equations are at the axiom level. Of course the two can happily co-exist.

Our framework for structured code generation is adopted from Meseguer's work on logic morphisms [Meseguer 89].

The direct impetus to the development of SPECWARE came from the desire to integrate several systems developed at Kestrel Institute over the last ten years, and the realization that they shared a common conceptual basis. These include the algorithm design system KIDS [Smith 90], the data type refinement system DTRE [Blaine and Goldberg 91], REACTO, a system for the development of reactive systems [Gilham et al. 89], and a synthesis system for visual presentations [Green 87]. An overview is presented in [Jüllig 93].

7 Conclusions

7.1 Summary

We presented the specification and refinement concepts of SPECWARE, a system aimed at supporting the application of formal methods to system development. Specware draws on theoretical work in formal specification and program synthesis as well as on experience with experimental systems over the past two decades. The development of SPECWARE continues; however, all concepts introduced here have been implemented. We have found the co-development of theory and implementation mutually beneficial.

The basic specification concepts of SPECWARE are specifications, specification morphisms, and diagrams of specifications and specification morphisms. The colimit operation takes diagrams of specifications to specifications.

The basic refinement notion is an interpretation, a morphism from a source specification into a definitional extension of a target specification. Interpretations are closed under sequential composition. To arrive at a notion of parallel refinement composition, we first observed that colimits and definitional extensions generate a Grothendieck topology on the category of specifications and specification morphisms, and that refinements form a sheaf with respect to this topology. Essentially this means that that given a specification diagram and an assignment of an interpretation to each node in the diagram one can construct an interpretation for the colimit of the given specification diagram, provided the compatibility condition holds: the interpretations assigned to the nodes must agree on shared parts.

The difficulty of checking the compatibility condition, among other reasons, prevented the direct application of this theory in practice. We instead developed diagram refinements as a practical realization; in diagram refinements the compatibility of interpretations is explicity ensured by the presence of interpretation morphisms.

7.2 Future Work

Current work includes adding to SPECWARE parameterized specifications and interpretations of parameterized specifications. This will lead to a vertical composition similar to that of Sannella's and Tarlecki's [Sannella and Tarlecki 88b] but to a different horizontal composition notion.

With the addition of parameterized specifications SPECWARE contains a set of primitives rich enough to allow for substantial experimentation. For this purpose we will recreate the algorithm design capabilities of KIDS in SPECWARE. We also expect the addition of code generation to other programming languages in addition to LISP.

A The Logic of Slang

The specification language used in SPECWARE is called SLANG. We distinguish SLANG because SPECWARE may have multiple back-ends, Lisp, C, Ada, etc., each with its own logic.

SLANG is based on higher-order logic, or higher-order type theory, as described in [Lambek and Scott 86]. However, unlike Lambek and Scott, we use classical logic (rather than intuitionistic logic) because the theorem prover currently used in SPECWARE is a resolution prover based on classical first-order logic (with some higher-order facilities).

Logically speaking, a SLANG specification is a finite collection of sorts, operations, and theorems (some of which are axioms). For pragmatic reasons, we have added sort-axioms (which are currently used to name sort terms), constructor sets (which are equivalent to induction axioms), and definitions (which are sets of axioms characterizing new operation symbols).

Every SLANG specification can be freely completed to a topos (see [Lambek and Scott 86, Section II.12] for a description of this construction). The objects in this topos are all sorts definable in the specification; the arrows are all definable operations (i.e., provably functional relations).

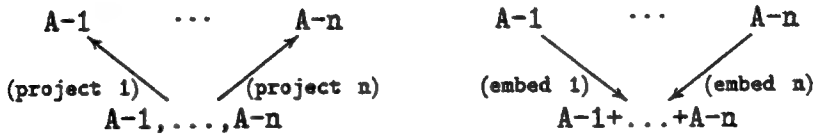
Built-in Constructs

The only sort which is built-in, i.e., is implicitly part of every specification, is `Boolean`. Along with this sort, the standard operations on it such as `true`, `false`, `and`, `or`, etc., and axioms characterizing them are built-in. The universal (`fa`) and existential (`ex`) quantifiers, and a polymorphic equality (`equal`) are also built-in.

Sort Constructors

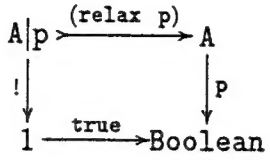
Lambek and Scott adopt a minimal set of sort constructors. While this is theoretically economical, we have chosen a richer set of sort constructors which arise in practice, especially in interpretations. We will use the generated topos to characterize these sort constructors; it is straightforward to generate the corresponding axioms.

N-ary products and coproducts. Given a set of n sorts, their product and coproduct are sorts which come equipped with the normal projections and embeddings, and characterized by the usual universal property.

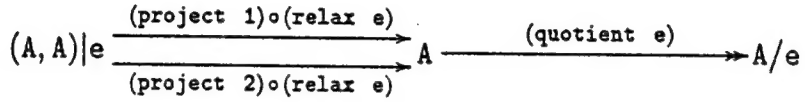


Function sorts. Given two sorts `A` and `B`, the function sort from `A` to `B`, written `A -> B`, satisfies the usual universal property and comes equipped with an evaluation operation, written `(<rator> <ap>)`, and an abstraction operation, written `(lambda (<args>) <body>)`.

Subsorts. Given a sort `A` and a predicate `op p: A -> Boolean` on this sort, the subsort of `A` consisting of those elements which satisfy the predicate, written `A|p`, and the induced injection are characterized by the following pullback diagram (1 is the terminal object, and ! denotes the unique arrow into it from `A|p`).



Quotient sorts. Given a sort `A` and an equivalence relation `op e:A,A -> Boolean` on this sort, the quotient sort consisting of equivalence classes of elements of `A`, written `A/e`, and the induced surjection are characterized by the following coequalizer diagram (`(A,A)|e` is the equivalence relation as a subsort of `A,A`).



Sort Axioms

Sort axioms are equations between sorts. Currently, these are restricted so that the left-hand side is a primitive sort (i.e., a sort which is not constructed using one of the sort constructors). Thus, in effect, sort axioms create new names for sorts. This keeps the sort algebra free, which is convenient for the type-checker. In the future, we may allow non-free sort algebras, and extend the type-checker to handle this.

Constructor Sets

A constructor set for a sort is a finite set of operations with that sort as the codomain. A constructor set is equivalent to an induction axiom. Here is an example.

```
constructors {zero, one, plus} construct NAT

axiom induction-for-NAT is
(fa (P) (implies
         (and (and (P zero) (P one))
              (fa (x y) (implies (and (P x) (P y))
                                 (P (plus x y)))))
         (fa (n) (P n))))
```

Note that a constructor set need not freely generate the constructed sort, i.e., the images of the constructors need not be disjoint. Additional axioms are necessary to force this.

Definitions

Definitions in SLANG are finite sets of axioms which completely characterize an operation. What this means is that to define a new operation `f: A -> B` in a specification S, there must be a formula `phi` with exactly two free variables `x:A` and `y:B` such that the relation specified by `phi` is provably functional in S:

```
(and (fa (x) (ex (y) (phi x y)))
     (fa (x) (implies (and (phi x y1) (phi x y2))
                      (equal y1 y2))))
```

Then S can be extended with the operation `f` together with the defining axiom

```
(iff (equal (f x) y) (phi x y)).
```

References

[Artin et al. 72]

ARTIN, M., GROTHENDIECK, A., AND VERDIER, J. L. *Théorie des Topos et Cohomologie Etale des Schémas, Lecture Notes in Mathematics,* Vol. 269. Springer-Verlag, 1972. SGA4, Séminaire de Géométrie Algébrique du Bois-Marie, 1963–1964.

[Bauer et al. 85]

BAUER, F. L., ET AL. *The Munich Project CIP, Volume I: The Wide Spectrum Language CIP-L, Lecture Notes in Computer Science,* Vol. 183. Springer-Verlag, Berlin, 1985.

[Bauer et al. 87]

BAUER, F. L., EHLER, H., HORSCH, A., MÖLLER, B., PARTSCH, H., PAUKNER, O., AND PEPPER, P. *The Munich Project CIP, Volume II: The Program Transformation System CIP-S, Lecture Notes in Computer Science,* Vol. 292. Springer-Verlag, Berlin, 1987.

[Bird 86]

BIRD, R. S. Introduction to the theory of lists. Tech. Rep. PRG-56, Oxford University Computing Laboratory, Programming Research Group, October 1986. Appeared in Logic of Programming and Calculi of Discrete Design, M. Broy, Ed., Springer-Verlag, NATO ASI Series F: Computer and Systems Sciences, Vol. 36, 1987.

[Bird 87]

BIRD, R. A calculus of functions for program derivation. Tech. Rep. PRG-64, Oxford University, Programming Research Group, December 1987.

[Blaine and Goldberg 91]

BLAINE, L., AND GOLDBERG, A. DTRE – a semi-automatic transformation system. In *Constructing Programs from Specifications*, B. Möller, Ed. North-Holland, Amsterdam, 1991, pp. 165–204.

[Burstall and Goguen 77]

BURSTALL, R. M., AND GOGUEN, J. A. Putting theories together to make specifications. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence* (Cambridge, MA, August 22–25, 1977), IJCAI, pp. 1045–1058.

[Gilham et al. 89]

GILHAM, L.-M., GOLDBERG, A., AND WANG, T. C. Toward reliable reactive systems. In *Proceedings of the 5th International Workshop on Software Specification and Design* (Pittsburgh, PA, May 1989).

[Goguen and Burstall 80]

GOGUEN, J. A., AND BURSTALL, R. M. CAT, A system for the correct elaboration of correct programs from structured specifications. Tech. Rep. CSL-118, SRI International, Oct. 1980.

[Goguen and Winkler 88]

GOGUEN, J. A., AND WINKLER, T. Introducing OBJ3. Tech. Rep. SRI-CSL-88-09, SRI International, Menlo Park, California, 1988.

[Green 87]

GREEN, C. Synthesis of graphical displays for tabular data. Tech. Rep. SBIR.FR.86.1, Kestrel Institute, October 1987. Final Report for Phase I; Note: accompanying videotape.

[Hoare 72]

HOARE, C. A. R. Proof of correctness of data representation. *Acta Informatica 1* (1972), 271–281.

[Jones 86]

JONES, C. B. *Systematic Software Development Using VDM.* Prentice-Hall, Englewood Cliffs, NJ, 1986.

[Jüllig 93]

JÜLLIG, R. Applying formal software synthesis. *IEEE Software 10*, 3 (May 1993), 11–22. (also Technical Report KES.U.93.1, Kestrel Institute, May 1993).

[Knuth 68]

KNUTH, D. E. *The Art of Computer Programming, Volume 1: Fundamental Algorithms.* Addison-Wesley, Reading, Massachusetts, 1968.

[Lambek and Scott 86]

LAMBEK, J., AND SCOTT, P. J. *Introduction to Higher Order Categorical Logic.* Cambridge University Press, Cambridge, 1986.

[Lehman et al. 84]

LEHMAN, M. M., STENNING, V., AND TURSKI, W. M. Another look at software design methodology. *ACM SIGSOFT Software Engineering Notes 9*, 2 (April 1984), 38–53.

[Mac Lane 71]

MAC LANE, S. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[Mac Lane and Moerdijk 92]

MAC LANE, S., AND MOERDIJK, I. *Sheaves in Geometry and Logic.* Springer-Verlag, New York, 1992.

[Meseguer 89]

MESEGUER, J. General logics. In *Logic Colloquium'87*, H.-D. Ebbinghaus et al., Eds. North-Holland, 1989, pp. 275–329.

[Sannella and Tarlecki 88a]

SANNELLA, D., AND TARLECKI, A. Specifications in an arbitrary institution. *Inf. and Comput. 76* (1988), 165–210.

[Sannella and Tarlecki 88b]

SANNELLA, D., AND TARLECKI, A. Toward formal development of programs from algebraic specifications: Implementations revisited. *Acta Informatica 25*, 3 (1988), 233–281.

[Smith 90]

SMITH, D. R. KIDS – a semi-automatic program development system. *IEEE Transactions on Software Engineering Special Issue on Formal Methods in Software Engineering 16*, 9 (September 1990), 1024–1043.

[Smith 93]

SMITH, D. R. Constructing specification morphisms. *Journal of Symbolic Computation, Special Issue on Automatic Programming 15*, 5-6 (May-June 1993), 571–606.

[Smith and Lowry 90]

SMITH, D. R., AND LOWRY, M. R. Algorithm theories and design tactics. *Science of Computer Programming 14*, 2-3 (October 1990), 305–321.

[Srinivas 93]

SRINIVAS, Y. V. A sheaf-theoretic approach to pattern matching and related problems. *Theoretical Comput. Sci. 112* (1993), 53–97.

[Turski and Maibaum 87]

TURSKI, W. M., AND MAIBAUM, T. E. *The Specification of Computer Programs.* Addison-Wesley, Wokingham, England, 1987.

[Wirsing 86]

WIRSING, M. Structured algebraic specifications: A kernel language. *Theoretical Comput. Sci. 42* (1986), 123–249. A slight revision of his Habilitationsschrift, Technische Universität München, 1983.

*U.S. GOVERNMENT PRINTING OFFICE: 1998-610-130-61164